TURING 图灵程序设计丛书

Programmation efficace

高效算法

竞赛、应试与提高必修128例

[法] Christoph Dürr Jill-Jênn Vie —— 著

史世强 —— 译

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

作者简介

Christoph Dürr，法国国家科学研究院研究员，巴黎皮埃尔-玛丽·居里大学博士生导师，Operation Research科研组研究主任。

Jill-Jênn Vie，法国高等电力学院博士、算法讲师，担任法国高等师范学院Paris-Saclay团队在ACM竞赛中的算法导师；曾任法国国际编程大赛Prologin主席，并于2014年获Google RISE Award。

译者简介

史世强

网名jetwaves，毕业于华中科技大学，法国特鲁瓦技术大学硕士，全栈工程师，曾在法国Aerow SAS任技术负责人，花果山水果品牌联合创始人。从初中起开始参加信息学奥林匹克竞赛，热衷于软件架构、软件工程和传统行业信息化建设，目前关注人工智能领域。白云黄鹤幽默版版主，平时活跃于知乎，同时也是野战游戏和健身运动爱好者。

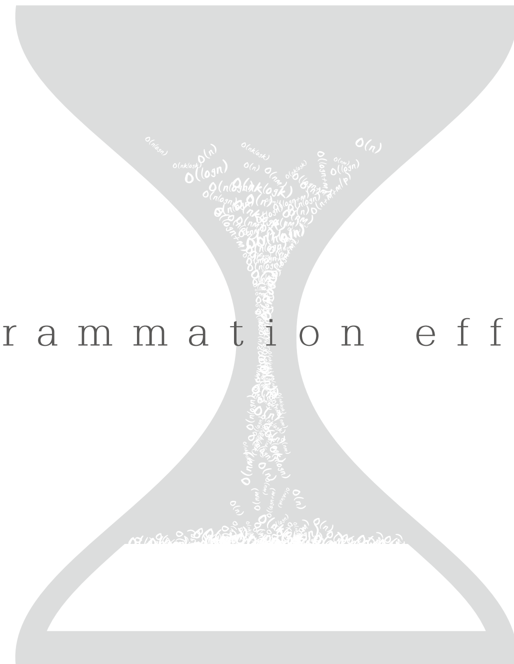
数字版权声明

图灵程序设计丛书

高效算法

竞赛、应试与提高必修128例

[法] Christoph Dürr　Jill-Jênn Vie —— 著
史世强 —— 译



Programmation efficace

人 民 邮 电 出 版 社
北　京

图书在版编目（CIP）数据

高效算法：竞赛、应试与提高必修128例 /（法）克里斯托弗·杜尔（Christoph Dürr），（法）吉尔-让·维（Jill-Jênn Vie）著；史世强译. -- 北京：人民邮电出版社，2018.5

（图灵程序设计丛书）

ISBN 978-7-115-48085-9

Ⅰ. ①高… Ⅱ. ①克… ②吉… ③史… Ⅲ. ①计算机算法—研究 Ⅳ. ①TP301.6

中国版本图书馆CIP数据核字（2018）第050544号

内 容 提 要

本书旨在探讨如何优化算法效率，详细阐述了经典算法和特殊算法的实现、应用技巧和复杂度验证过程，内容由浅入深，能帮助读者快速掌握复杂度适当、正确率高的高效编程方法以及自检、自测技巧，是参加ACM/ICPC、Google Code Jam等国际编程竞赛、备战编程考试、提高编程效率、优化编程方法的参考书目。

◆ 著　　[法] Christoph Dürr　Jill-Jênn Vie
译　　史世强
责任编辑　戴　童
责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：12.75
字数：301千字　　2018年5月第1版
印数：1-3 500册　　2018年5月北京第1次印刷

著作权合同登记号　图字：01-2017-3131号

定价：55.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字20170147号

译者序

22 年前的秋天，我刚刚进入初中时，得到了一台中华学习机。它的 1 MHz 主频甚至赶不上现在一台 10 元钱的计算器。我从第一行用 BASIC 语言写的 IF/ELSE 开始，开启了自己的编程人生。1996 年，还是初中生的我凭着不多的算法和逻辑知识参加了国家信息学奥林匹克竞赛，当然，最后只得到了安慰奖。二十多年后，我得知当年斩获金牌的是王小川，如今搜狗的 CEO。

现在，我在一家互联网公司负责技术并管理研发团队。从自身的职业发展经历，以及在中国和法国的招聘和用人经历中，我深刻体会到了软件工程师的成就在很大程度上取决于他的专业知识视野。这是个很现实的问题。因此，我在得到翻译这本法语技术书的机会时，欣然接受了这个颇有难度的任务。

法国是一个盛产数学家的国度。不同于大家的传统印象，法国人在“浪漫”的同时，在工作和科研中非常讲究逻辑与验证——产品原型要验证，技术探索要验证。证明和实验有着同样不可或缺的地位。理论和实践的结合，让法国学界和企业界在相当长时间内保持着旺盛的生命力与创造力。这是我在法国 8 年学习和工作中的真实体验。

本书由法国国际信息学奥林匹克竞赛“国家队”辅导老师编写，凝聚了作者辅导高中生、大学生参加国际信息学奥林匹克竞赛的大量经验和技巧。书中提及的部分算法十分常见，在实际工作中也十分常用。但也有另一部分算法，例如舞蹈链算法以及一些涉及图论与匹配的算法，在中国的大学教育都不太提及。

在人工智能和深度学习大发展的今天，Python 语言、算法，特别是证明算法可靠性和高效性的能力，是进入大数据和人工智能人才市场的入场券。希望读者善用 Github 和作者准备的源代码网站，以及网上能够找到的技术资源，在尝试代码实现的同时，去理解算法复杂度的证明过程，从而彻底掌握并熟练运用这些凝聚了很多代人智慧的无形资产。

我要感谢教我写下第一行代码的哥哥史轶，支持并指导我参加国家信息学奥林匹克竞赛的湖北省十堰市东风汽车公司第四中学的陈长国老师，用大量课外知识开拓了我的见识的东风汽车公司第一中学的吴华山老师，支持我前往法国留学的父母，以及一直以来给我带来太多快乐的妻子和孩子。

由于个人水平有限，译文不能做到尽善尽美，欢迎读者通过我的个人网站 www.jetwaves.cn 与我交流。

史世强

2017 年 10 月于巴黎

序

我们编写本书的主要动力是对 Python 语言编程的热爱和对解决算法问题的激情。Python 语言能够如此打动人，是因为这种语言能让我们编写清晰而优雅的代码，把注意力集中于算法的本质步骤，而不需要过多关注复杂的语法和数据结构。同时，我们用 Python 完成编写程序后数个月再回头来读的时候，仍然可以理解自己写的代码，这一点十分有教益。作为本书的作者，我们最希望的是能接受新的挑战，其次是能经得住各种测试，因为一段程序代码只有在毫无 bug 地实现后，我们才算真正地掌握了编程技巧。我们希望用自己的热情感染读者，营造出一种氛围，鼓励大家学习和掌握扎实的算法和编程基础知识。这种学习经历往往会受到大型软件企业招聘人员的赏识，而对于软件工程师或计算机科学教育工作者来说，这对其整个职业生涯也会有所帮助。

本书按照主题而不是技术分类收录了 128 种算法。其中某些算法是常见的经典算法，另一些则不太常见。尤其在读者备战 ACM-ICPC、Google Code Jam、Facebook Hacker Cup、Prologin 和 France-ioi 等编程竞赛时，本书编写的大量问题将起到积极的辅导作用。我们希望本书能够成为算法的基础教程和高级程序设计教程的参考，或者能让学习数学和计算机专业的读者看到与众不同的进修内容。读者可以在网站 tryalgo.org（http://tryalgo.org/code/）上找到本书使用的源代码库①, 以及用来测试代码调试结果和实现性能的链接。

感谢 Huong 和智子，如果没有这两位朋友的支持，本书是无法完成的。感谢法国综合理工学院和法国高等师范学院 Cachan 分校的学生们，他们多次通宵达旦的训练，为本书提供了很多素材。最后，感谢所有审阅手稿的朋友们，他们是 René Adad、Evripidis Bampis、Binh-Minh Bui-Xuan、Stéphane Henriot、Lê Thành Dũng Nguyễn、Alexandre Nolin 和 Antoine Pietri。本书的作者之一要特别感谢在 Tiers 高中时的老师 Yves Lemaire 先生：当年就是在这位老师的启迪下，作者才初次发现了本书 2.5 节中描述的“宝藏”。

最后，我们希望读者在碰到算法难题时，能够耐心地花时间去思考。祝愿大家能在豁然间找到解答，甚至是一个优雅的解答，享受到胜利的喜悦之情。

好，我们要开始了！

---

① 也可以用 PyPI 直接安装后下载查看并执行。——译者注

目录

第1章 引言

年轻人，通过本书学习编写算法，你将在编程竞赛中大显身手，顺利通过就业面试，卷起袖管大干一场，创造更多的价值。

如今人们仍然存在一种误解，错把程序员当成当代的魔术师。计算机逐渐进入企业和家庭，成为推动世界运行的重要动力。但是，仍有太多人在使用计算机的时候没能掌握足够的知识，充分发挥计算机的能力，来满足自己的需要。懂得编程可以让人们在最大程度上找到解决问题的高效方法。算法和编程成为计算机行业中必不可少的工具。掌握这些技能可以让我们在面对困难时提出有创造力、高效的解决方案。

本书介绍了多种解决某些经典问题的算法技术，描述了问题出现的场景，并用 Python 提出了简单的解决方案。正确地实现算法往往不是一件简单的事情，总需要避开陷阱，也需要应用一些技巧保证算法能够在规定时间内实现。本书在阐述算法实现时附加了重要的细节，以帮助读者理解。

最近几十年，不同级别的编程竞赛在世界范围内展开，推广了算法文化。竞赛考察的问题一般都是经典问题的变种，隐藏在难以破解的谜面背后，让参赛者们一筹莫展。

1.1 编程竞赛

在编程竞赛中，参赛者必须在规定时间内解决多个问题。问题的输入称为实例（instance）。举个例子，一个输入实例可以是最短路径问题中图的邻接矩阵。一般来讲，问题会给出一个输入实例和它的输出结果[①]。参赛者在网上将答案的源代码提交到服务器；之后，服务器的后台进程将编译并

① 用于展示思路和代码测试。——译者注

执行代码，而后测试对错。对于某些问题，源代码在执行时会被输入多个实例，并一一执行；而对于其他问题，每次执行源代码时，输入都从一个表示实例数量的整数开始。程序必须按顺序读取每个输入实例，解决问题，并输出结果。如果程序能够在指定时间内输出正确结果，那么提交的答案就可以被接受。

图 1.1　ACM 竞赛的图标形象地展示了解决问题的步骤。参赛团队每解决一个问题时，就会得到一个吹起的气球

我们无法列出世上所有的编程竞赛名称和竞赛网址。就算有可能，这个列表也会很快过时。但无论如何，我们在这里还是要简单介绍一下最重要的几个编程竞赛。

- ACM/ ICPC 编程竞赛

这是历史最悠久的竞赛，由国际计算机协会（ACM）从 1977 年开始举办。竞赛称为“国际大学生程序设计竞赛”（ICPC），以巡回赛的方式进行。比如，巡回赛在法国站的起点是西南欧洲地区竞赛（SWERC）。地区竞赛的前两名有资格进入全球决赛。这个竞赛的特点是每队由 3 位成员组成，共用一台计算机。参赛队在 5 个小时内从 10 个问题中尝试挑战解决尽可能多的问题。排名的第一个依据是答案被接受的数量（答案会被不公开的用例来测试）；排名的第二个依据是解决问题所耗费的时间，耗时以开始解题到提交答案的时长为准。提交一个错误答案会被罚时 20 分钟。

组成一个优秀团队有很多种方式。一般来说，至少需要一位优秀的程序员和一位优秀的数学家，以及一位擅长不同领域的专家，比如图论、动态规划等。他们需要在承受巨大压力的前提下通力合作。在竞赛中，参赛者可以用 8 磅字体打印 25 页的源代码作为参考。参赛者还可以访问 Java 应用程序编程接口（API）的在线文档，以及 C++ 的在线标准库文档。

- Google Code Jam 编程竞赛

国际计算机协会的编程竞赛仅限硕士及以下学历的学生参加，与此不同的是，Google Code Jam 编程竞赛对所有人开放。竞赛每年一度，举办历史较短，而且仅限个人参赛。每个问题通常会包含一系列简单实例，解答这些实例就可以得到一定的分数。同时，问题还包含一系列步骤复杂的实例，这需要真正找到拥有合适复杂度的算法来解决。直到竞赛结束，参赛者才能得知步骤较复杂的实例是否最终被接受了。这个竞赛的优势在于，参赛者在竞赛结束后可以查阅其他参赛者提交的解决方案，这种方式有非常强的指导作用。Facebook Hacker Cup 编程竞赛也采取类似形式。

- Prologin 编程竞赛

法国每年为 20 岁以下的学生举办一场 Prologin 编程竞赛。竞赛过程分为三步——在线筛选、

地区赛和决赛，以考察参赛者解决算法问题的能力。最终决赛是一场不同寻常的 36 小时竞赛，参赛者需要解决一个人工智能问题。每个参赛者必须编写一个遵循组织者设定规则的游戏程序，然后以循环赛的形式让游戏程序彼此对决，以此来决定参赛者的成绩排名。竞赛官网上 prologin.org 对此有详尽的解释，我们也可以在这里测试自己的算法。

- **France-ioi 编程竞赛**

France-ioi 协会旨在辅助法国初中生和高中生准备国际信息学奥林匹克竞赛。从 2012 年起，协会每年举办"河狸计算机科学竞赛"（竞赛的吉祥物是一只河狸），从初中一年级到高中三年级的学生均可参加。2014 年，全法国有 22.8 万名参赛者。协会官网 france-ioi.org 汇集了 1000 多个有代表性的现象级算法题。

除了上述竞赛以外，也有大量以筛选求职者为目的举行的编程竞赛。比如 TopCoder 网站不仅进行测试，也会对算法进行详细解释，有时讲解质量极高。如果读者希望训练编程能力，我们特别推荐 Codeforces，这是一个备受竞赛群体推崇的网站，对问题的解释总是清晰而仔细。

1.1.1　线上学习网站

很多网站提供历年各大竞赛真题，并可在线测试答案，供大家学习训练。Google Code Jam 编程竞赛和 Prologin 编程竞赛的官网也提供此类功能。但是，ACM/ICPC 每年的竞赛题目却没有统一归纳。

- **传统的线上训练和裁判网站**

下列网站 uva.onlinejudge.org、icpcarchive.ecs.baylor.edu 和 livearchive.onlinejudge.org 总结了大量的 ACM/ICPC 编程竞赛的试题和答案。

- **中国的线上训练和裁判网站**

中国目前有很多线上训练算法能力的网站，比如北京大学的 poj.org、天津大学的 acm.tju.edu.cn 和浙江大学的 acm.zju.edu.cn。相对于其他网站，这些网站更注重训练功能。

- **高级语言算法的训练和裁判网站**

spoj.com（Sphere Online Judge）网站接受用户使用更多种编程语言提交问题的解决方案，其中包括 Python。

在本书配套网站 tryalgo.com 中，读者可以找到应用本书各章讲解的知识和技巧来解决的问题，在实践中检验从书中学到的算法知识。

编程竞赛主要使用的编程语言是 C++ 和 Java 语言。Google Code Jam 编程竞赛接受所有编程语言，因为解题过程是参赛者在本地开发环境中完成的。除此之外，上面提到的线上训练和裁判网站 SPOJ 也接受 Python 语言的解答方案。为了解决因编程语言不同而导致的程序执行时间的差异问题，线上训练和裁判网站对使用不同编程语言的解答方案给出了不同的时间限制。但是，这种平衡策略并不总是准确的，而且，用 Python 语言完成的解题方案经常不能被正确执行。我们希望这种

情况在未来几年能够有所改善。某些线上训练和裁判网站仍在使用 Java 语言的老版本，导致有些很实用的**类**无法使用，如 Scanner 类。读者在使用这些网站的时候，应当注意版本兼容问题。

1.1.2 线上裁判的返回值

当一段代码被提交给线上训练和裁判网站的时候，会被一系列不公开的测试用例测试，测试结果和一段简要的返回值会反馈给提交者。返回代码有以下几种。

- **Accepted：已接受状态**

  你提交的代码在指定时间内给出了正确的结果，祝贺你！

- **Presentation Error：展示错误**

  程序基本能够被接受，但显示了过多或过少的空格或换行符。这种返回码很少出现。

- **Compilation Error ：编译错误**

  你的程序在编译过程中出了错。一般来说，当你点击这条返回码的时候，就能得到错误的详细信息。你应当比较一下，裁判和自己使用的编译器版本是否有所不同。

- **Wrong Answer：错误答案**

  重新读一遍题吧，你肯定漏掉了什么细节。你是否确定已经检查了所有的边界条件？你是否在代码中遗留了调试代码？

- **Time Limit Exceeded：执行超时**

  你的解答方案可能没有达到足够优化的实现效率，或者代码的某个角落里藏着一个死循环。检查循环变量，确保循环能够终止。使用一个大规模的复杂测试用例在本地执行测试，确保你的代码性能。

- **Runtime Error：运行时错误**

  一般来说，这种错误源于分母为 0 的除法运算、数组下标越界，或者对一个空的堆执行了 `pop()` 方法。其他情况也会产生这条错误提示，比如在使用 Java 语言的解答方案中使用了 `assert` 断言，这种方式在编程竞赛中一般是不被接受的。

除了以上有明确意义的返回代码，没有返回代码的情况也能够或多或少地提供一些信息，帮助查找错误。以下是一个 ACM/ICPC/SWERC 竞赛中的真实案例。在一道关于图的题目中，明确指出了输入数据是连通图，但某个参赛团队对此信息不太确定，于是编写了一个测试连通性的方法：当这个方法返回 `true` 结果，即输入为连通图时，程序会进入死循环（返回执行超时错误）；而当这个方法返回 `false` 结果，即输入为非连通图时，程序会执行一个分母为 0 的除法（返回运行时错误）。这种方法可以帮助参赛者探测到某些测试用例输入的图并不是题目中的连通图，从而避免错误。[①]

① 这种方法的目的是在程序中故意留一些缺陷，从而通过返回值来猜测输入数据的具体情况。
——译者注

1.2　我们的选择：Python

鉴于 Python 编程语言的可读性和使用的简易性，本书选用它来描述算法。在工业领域，Python 通常用于制作程序的原型。Python 也用于如 SAGE 这类重要的项目系统，因为其中的核心内容大多用实现速度快很多的语言编写，如 C 或 C++。

现在我们说说 Python 编程语言的一些细节。在 Python 中有四个基本数据类型：布尔型、整型、浮点型和字符串。与其他大多数的编程语言不同，Python 中的整数不受数字占用的二进制位数限制，而使用高精度计算方式。

Python 中的高级数据类型包括字典（dictionary）、列表（list）和元组（tuple）。列表和元组的区别是，元组是不可变数据，因此可以用作字典中键值对数据的键。

网络上有很多 Python 的入门教程，如官网 python.org。David Eppstein 创建了一个名为“Python 算法和数据结构”（PADS）的元件库，其中也有很好的讲解。

在编写本书代码的过程中，我们遵循了 PEP8 规范。该规范细致地规定了空格的使用方法、变量命名规则，等等。我们建议读者也遵循上述规范。

- Python 2 还是 Python 3？

Python 3.x 版本已经于 2008 年发布。但直到今天，由于仍有大量的类库没有迁移到 Python 3.x，使得许多开发工作还继续停留在 Python 2.x 版本。尽管如此，我们仍然选择使用 Python 3.x 来实现算法。Python 2.x 和 Python 3.x 对本书中代码的主要影响在于 print 语句的使用方式，以及整数除法的使用方式。在 Python 3.x 中，对于两个整数 *a* 和 *b*，表达式 *a*/*b* 会返回除法的浮点型的商，表达式 *a*//*b* 返回的则是两者的欧几里得商，即商的整数部分。`print` 的用法区别在于，在 Python 2.x 中 `print` 是语句，而在 Python 3.x 中 `print()` 是需要使用括号包围的参数来调用的函数。

如果程序运行存在性能问题，可以考虑使用 pypy 或 pypy3 解释器来执行，因为这都是实时编译器。也就是说，Python 代码会先被翻译为机器码，然后才被干净而迅速执行。但 pypy 的弱点在于它仍处在开发过程中，很多 Python 类库尚无法支持。

- 无穷

Python 使用高精度计算方式进行计算，而不用二进制位数来限制整数的大小。所以，在 Python 语言中不存在哪个数可以指代正无穷大或负无穷大的值。但对于浮点数，我们可以用 `float('inf')` 和 `float('-inf')` 来指代正、负无穷大。

- 一些建议

Python 的初学者在复制列表数据时经常犯一个错误。在下面的例子里，列表 B 只是一个指向列表 A 的引用。对 B[0] 的修改同样会修改 A[0]。

```
A = [1, 2, 3]
B = A
```

当复制一个 A 的独立副本时，我们可以使用以下语法格式：

```
A = [1, 2, 3]
B = A[:]
```

语句 `[:]` 用以复制一个列表。我们也可以复制一个去掉首元素的列表 A[1:]，或者去掉末尾元素的列表 A[:-1]，或者逆序的列表 A[::-1]。举例来说，下面的代码会生成一个所有行完全相同的矩阵 M，而对 M[0][0] 元素的修改会导致第一列所有元素被修改。

```
M = [[0] * 10] * 10
```

我们可以用下面两种正确的方式来初始化一个这样的矩阵：

```
M1 = [[0] * 10 for _ in range(10)]
M2 = [[0 for j in range(10)] for i in range(10)]
```

操作矩阵的简单方式是使用 numpy 模块，但我们在本书中不使用第三方类库，以便让程序代码能更方便翻译成 Java 或 C++ 代码。

另一个典型错误经常发生在使用 `range` 语句时。比如，下面的代码会顺序处理列表 A 中 0 至 9 号元素（包括 0 号和 9 号元素）：

```
for i in range(0, 10):            # 包括 0，不包括 10
    treat(A[i])
```

如果想逆序处理上述元素，仅反转参数是不够的。语句 `range(10, 0, -1)` 中的第三个参数代表循环的步长，语句会导致被处理元素中的 10 号元素被包含在内，而 0 号元素被排除在外。因此需要用以下方式来处理：

```
for i in range(9, -1, -1):        # 包括 9，不包括 -1
    treat(A[i])
```

1.3 输入输出

1.3.1 读取标准输入

在大部分编程竞赛的题目中，源数据都需要从标准输入设备来读取，并把输出显示到标准输出设备上。如果输入文件名叫 `test.in`，你的程序名叫 `prog.py`，那就可以在控制台执行以下命令，将输入文件的内容重定向到你的程序：

```
python prog.py < test.in
```

一般来说，在 Mac OS X 系统中，控制台可以用 Command + 空格，呼出 SpotLight 搜索后键入 Terminal 来打开；在 windows 系统中，使用“开始 – 执行 – cmd”；在 Linux 系统中，使用快捷键 Alt-F2[①]。

如果你想把程序的输出记录到名为 `test.out` 的输出文件中，使用的命令格式如下：

```
python prog.py < test.in > test.out
```

小技巧，如果你想把输出写入文件 `test.out`，同时还要显示在屏幕上，可以使用以下命令（注意，`tee` 命令在 Windows 环境下默认是不存在的）：

```
python prog.py < test.in | tee test.out
```

输入数据文件可以使用 `input()` 语句按行读取。`input()` 语句把读取到的行用字符串的形式返回，但不会返回行尾的换行符[②]。在 sys 模块中有一个类似的方法 `stdin.readline()`，这个方法不会删除行尾的换行符，但根据我们的经验，它的执行速度是 `input()` 语句的 4 倍。

如果读取到的行包含的应当是一个整数，我们使用 `int` 方法进行类型转换；如果是一个浮点数，我们使用 `float` 方法。当一行中包含多个空格分隔的整数时，我们首先使用 `split()` 方法把这一行拆分成独立的部分，然后用 `map` 方法把它们全部转换成整数。举例来说，当用空格分隔的两个整数——高度和宽度，需要在同一行内被读取时，可以使用以下命令[③]：

```
import sys
height, width = map(int, sys.stdin.readline().split())
```

如果你的程序在读取数据时遇到性能问题，根据我们的经验，可以仅使用一次系统调用，把整个输入文件读入，速度即可提升 2 倍。在下列语句中，假设输入数据中只有来自多行输入的整数，`os.read()` 方法的参数 0 表示标准输入流，常量 M 必须是一个大于文件大小的限值。比如，文件中包含了 10^7 个大小在 0 至 10^9 之间的整数，那么每个整数最多只能有 10 个字符，而两个整数中间最多只有两个分隔符（\r 和 \n，即回车和换行），我们可以选择 $M = 12 \cdot 10^7$ 。

---

① 使用组合快捷键是个好习惯。在 Windows 环境下，Windows 7 和 windows XP 系统都可以使用上述方式，而在 Windows 8、Windows 10 和 Windows Server 环境下，建议使用 Windows+R 组合键呼出“运行命令”窗口，再输入 cmd 打开控制台，或在 Windows 8 和 Windows 10 环境下，直接按 Windows 键打开“开始”屏幕，输入 cmd 后回车，也可以快速打开控制台。——译者注

② 根据操作系统的不同，换行符可能是 \r 或 \n，或二者皆有，但使用 `input()` 输入的时候不需要考虑这个问题。注意在 Python2.x 中，`input()` 方法的行为是不同的，同样，应当使用等价的 `raw_input()` 方法。

③ 以下命令中使用了 `map` 及管道概念。——译者注

```
    import os
    inputs = list(map(int, os.read(0, M).split()))
```

- **例子：读取三个矩阵 A、B、C，并测试是否 AB = C**

在此例子中，输入格式如下：第一行包含一个唯一的整数 n，接下来的 $3n$ 行，每行包含 n 个被空格分隔的整数。这些行代表三个 $n \times n$ 矩阵 A、B、C 内包含的所有元素。例子的目的是测试矩阵 A × B 的结果是否等于矩阵 C。最简单的方法是使用矩阵乘法的解法，复杂度是 $O(n^3)$。但是，有一个可能的解法，复杂度仅有 $O(n^2)$，即随机选择一个向量 x，并测试 $A(Bx) = Cx$。这种测试方法叫作 Freivalds 比较算法（见参考文献 [8]）。那么，程序计算出的结果相等，而实际上 AB ≠ C 的概率有多大呢？如果计算以 d 为模，错误的最大概率是 $1/d$。这个概率在多次重复测试后可以变得极小。以下代码产生错误的概率已经低至 10^{-6} 量级。

```
from random import randint
from sys import stdin

def readint():
    return int(stdin.readline())

def readarray(typ):
    return list(map(typ, stdin.readline().split()))

def readmatrix(n):
    M = []
    for _ in range(n):
        row = readarray(int)
        assert len(row) == n
        M. append(row)
    return M

def mult(M, v):
    n = len(M)
    return[sum(M[i][j] * v[j] for j in range(n)) for i in range(n)]

def freivalds(A, B, C):
    n = len(A)
    x = [randint(0, 1000000) for j in range(n)]
    return mult(A, mult(B, x)) == mult(C, x)

if __name__ == " __main__ ":
    n = readint()
    A = readmatrix(n)
    B = readmatrix(n)
    C = readmatrix(n)
    print(freivalds(A, B, C))
```

1.3.2　显示格式

程序的输出必须使用 `print` 命令，它会根据你提供的参数生成一个新的行。行尾的换行符可以通过在参数中传递 `end=''` 取消掉。为显示指定小数位数的浮点数，可以使用 % 运算符，方法为“格式 % 值”。第 i 个占位符会被值列表中的第 i 个值替换。以下例子显示了一行格式类似“Case #1: 51.10 Paris”的字符串：

```
    print("Case #%i: %.02f %s" % (testCase, percentage, city))
```

在上面例子中，%i 被整型变量 `testCase` 的值所替换，%.02f 被浮点型变量 `percentage` 的值所替换并保留两位小数，%s 被字符串型变量 `city` 的值所替换。

1.4　复杂度

要想写出高效率的程序，必须先找到一个具有合适复杂度的算法。复杂度取决于运算时间和输入数据大小之间的关系。我们用朗道表达式（大 O 符号）来表示不同算法的复杂度。假设输入数据或参数的长度为 n，且算法的运算时间随 n^2 变化，那么我们就说这个算法的复杂度是 $O(n^2)$。

对于两个正值函数 f 和 g，如果存在正实数 n_0 和 c，对于所有 $n \geqslant n_0$ 都满足 $f(n) \leqslant c \cdot g(n)$，则我们借此定义函数之间的关系，并简记为 $f \in O(g)$。**由于滥用符号，也有人写做** $f = O(g)$。这种记法能够把函数 f 中的乘法常量和加法常量抽象出来，体现出函数运算时间相对于参数长度的增长速度。

同样，对于常量 n_0 和 c（$c > 0$），如果对于所有 $n \geqslant n_0$ 都能够满足 $f(n) \geqslant c \cdot g(n)$，则记作 $f \in \Omega(g)$。如果 $f \in O(g)$ 且 $f \in \Omega(g)$，则记作 $f \in \Theta(g)$，它表示 f 和 g 函数拥有相同的时间复杂度。

当 c 是一个常量且算法的复杂度是 $O(n^c)$ 的时候，我们说这个算法的复杂度和 n 成**多项式时间**关系。当一个问题存在一种算法解，而且解的复杂度是**多项式时间**的时候，该算法就是一个需要多项式时间解决的问题。这类问题有一个专门的名字叫作 P 问题[①]。遗憾的是，不是所有的问题都存在多项式时间解。还有大量问题，人们尚未找到任何能够在多项式时间内解决的算法。

其中一个问题是布尔可满足性问题（k-SAT）：给定 n 个布尔型变量和 m 条语句，每条语句包含 k 个符号（每个符号代表一个变量或其逆值变量），是否有可能为每个变量赋一个布尔值（**真**或**假**），使得每条语句包含至少一个值为**真**的变量？（SAT 是布尔可满足性问题中语句对符号数量没有限制的版本。）每一个单独问题[②]的特殊性在于，我们能够在多项式时间内通过评估所有条件，验证一个潜在的解（变量赋值）能否满足以上所有限制。当以上条件被满足的时候，这类问题有个专

① 在计算复杂度理论中，P 是在复杂度类问题中可于决定性图灵机以多项式量级或称多项式时间求解的决定性问题。——译者注

② k 取不同值的时候。——译者注

门的名字叫作 NP 问题①。我们可以很容易在多项式时间内解决 1-SAT，因此 1-SAT 问题属于 P 问题。2-SAT 同样也属于 P 问题，我们将在 6.10 节验证它。但从 3-SAT 开始，我们就不确定了。我们只知道解决 3-SAT 问题的难度至少和 SAT 问题的难度相当。

若恰好 P ⊂ NP，直观看来，如果我们能找到一个多项式时间复杂度的解，那就一定可以找到一个非定常多项式时间复杂度的解。人们认为 P ≠ NP，但目前这个推测仍然得不到证实。在证实之前，研究者们把 NP 问题简化，把问题 A 的多项式时间算法解转化为问题 B 的解。如此一来，如果 A 问题属于 P 类，那么 B 问题也同样属于 P 类——A 问题的难度和 B 问题的难度“至少是相同的”。至少和 SAT 难度相同的问题集合构成了一个问题的类别，即 NP 困难问题。它们中有一部分既是 NP 困难问题，又属于 NP 问题，那么这些问题则属于 NP 完全问题。无论是谁，只要能在多项式时间内解决其中一个问题，就可以解决所有其他问题。而这个人也会被历史铭记，同时得到一百万美元的奖金。目前，为了在可接受的时间内解决这些问题，挑战者必须专注于那些有助于解决问题的方向和领域（如图的平面性问题），或者让程序能用稳定的概率返回结果，或者提出接近最优解的解决方案。幸运的是，那些在编程竞赛中可能遇到的问题总体来说都是多项式时间复杂度问题的。②

在个人编程竞赛中，参赛者的程序必须在几秒钟内给出结果，这只留给处理器执行上千万或上亿次运算的时间。表 1.1 给出了针对不同的输入数据长度，以及在 1 秒钟内给出结果的算法的可接受时间复杂度标准。要注意，这些数字取决于编程语言③和执行程序的硬件设备，以及要执行的运算类型，如整数运算、浮点数运算或调用数学函数。

表 1.1

输入数据长度	可接受的复杂度
1000000	$O(n)$
100000	$O(n \log n)$
1000	$O(n^2)$

我们请读者用简单程序做一个实验，测试用不同的 n 值做 n 次乘法所需要的运算时间。我们坚持认为，在朗道表达式中，那些隐藏常量值也可能非常重要，而且有时在实践中，算法的渐进时间复杂度越大，就越有可能成功。举个例子，当计算两个 $n \times n$ 阶矩阵乘法的时候，贪婪算法需要 $O(n^3)$ 次运算，然而 Strassen 发现了一个只需要 $O(n^{2.81})$ 次运算的递归算法（见参考文献 [26]）。但对于实际要进行的矩阵运算，贪婪算法显然更加有效率。

在 Python 中，在列表中添加一个元素所需要的时间是一个常数，同样，访问一个指定下标

① 非定常多项式时间复杂性类，包含了可以在多项式时间内，对于一个判定性算法问题的实例，一个给定的解是否正确的算法问题。——译者注

② 如果读者对算法复杂度相关问题感兴趣，推荐阅读：《可能与不可能的边界：P/NP 问题趣史》，人民邮电出版社，2014 年。——编者注

③ 大致上，C++ 比 Java 语言快 2 倍时间，比 Python 快 4 倍时间。

的列表元素所需要的时间也是一个常数。新建一个列表的 L[i:j] 子列表所需要的时间是 $O(\max\{1, j-i\})$[①]。Python 语言中的字典型数据通过散列表（hash table）来表示和存储，在最坏情况下，访问一个键所需要的时间是线性的（由字典中键的数量决定），但实际上，访问时间一般是常数。然而，这个常数时间是不能忽略的，所以，如果字典的键值是 0 到 $n-1$ 的整数，最好使用列表性能。

对于某些数据结构，我们使用分摊时间复杂度。比如在 Python 中，一个列表在内部是用表格来展现的，并有一个**大小**属性。当用 `append` 方法将一个新元素加入列表的时候，它会被加入到表格的最后一个元素之后，列表大小属性加 1。如果表格的容量不足以添加新元素，则会分配一个内存空间是原表格大小 2 倍的新表格，并把原表格内容复制进来。同样，当对一个空列表连续执行 n 次 `append` 命令时，每次执行时间有时是常量，有时是与列表大小相关的线性值。但这些 `append` 方法的执行时间仍然在 $O(n)$ 级别，因为每次执行操作可以分摊一个 $O(1)$ 级别的常量时间。

1.5　抽象类型和基本数据结构

我们将首先讲述高效编程的核心内容——程序解决问题的基础，即数据结构。

抽象类型是关于一系列对象的规范，它归纳了对象可以取的值、可以执行的操作以及操作的具体内容。我们也可以把一个抽象类型理解为对象的统一规格。

数据结构是根据统一规格的定义，为高效处理特定数据而总结出的具体数据组织方式。因此，我们可以使用一个或多个数据结构来实现一个抽象类型，并设定每个操作的时间复杂度和所需内存。如此一来，根据操作被执行的频率，我们会选择某一种抽象类型的实现方式来解答不同问题。

为了更好地编写程序，必须掌握编程语言和标准库所提供的数据结构。在下面几节中，我们来讲解一下竞赛中最实用的数据结构。

1.5.1　栈

栈（stack）是把元素组织起来并提供如下操作的对象（图 1.2）：测试一个栈是否为空，在其顶部添加一个元素（入栈），从顶部访问并删除一个元素（出栈）。Python 语言的基本类型列表（list）实现了栈。我们使用 `append(element)` 方法执行入栈操作，使用 `pop()` 方法执行出栈操作。如果一个列表被用于布尔运算，比如一个 `if` 或 `while` 语句中的条件测试，语句当且仅当它非空的时候值为**真**。此外，其他所有实现了 `__len__` 方法的对象也是如此。以上所有操作需要的时间都是一个常数。

① 跟子列表本身的长度有关。——译者注

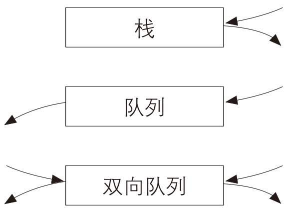


图 1.2 Python 语言中三种主要的访问序列数据结构

1.5.2 字典

字典能采用表格和下标的方式把键和值关联起来。其内部运行方式以散列表结构为基础；散列表结构使用散列算法把元素与表中的某个下标关联，并在多个元素与同一个下标关联的时候实现冲突处理机制。在最好的情况下，字典的读、写操作时间都是常数。但在最坏的情况下，所需时间是线性的，因为系统必须顺序访问一系列键和值，以便处理冲突[①]。在实际应用中，最坏的情况很少发生。在本书中，我们总体上都假设访问一个字典元素的时间是常数。如果键值的形式为 0, 1,⋯, *n*-1，我们通常建议使用简单的表结构而不是字典，令程序效率更高。

1.5.3 队列

队列与栈类似，差别仅在于向队列里添加元素时，元素被加到尾部（入队），而提取元素时则从队列头部开始（出队）。这种机制也称作 FIFO（first in, first out，先进先出），就像排队一样；而栈则被称作 LIFO（last in, first out，后进先出），就像垒一堆盘子一样。

在 Python 的标准库中，有两个类实现了队列。第一是 `Queue` 类，这是一个同步实现，意味着多个进程可以同时访问同一个对象。由于本书的代码不涉及并发机制，我们不推荐使用这个类，因为它在执行同步的时候使用的信号机制会拖慢执行速度。第二是 `Deque` 类（Double Ended Queue，即双向队列），除了提供标准方法，即在尾部使用 `append(element)` 添加元素和在头部使用 `popleft()` 提取元素之外，它还提供了额外方法，用于在队列头部使用 `appendleft(element)` 添加元素和在尾部使用 `pop()` 提取元素。我们把这种队列称作**双向队列**。这种更复杂的数据结构将在 8.2 节详细说明：在路径权重是 0 和 1 的图中查找最短路径算法中，这种结构非常有用。

我们推荐使用 `Deque` 类。但为了举例说明，以下代码展示了如何使用两个栈实现一个队列的方式。一个栈作为队列头部，用于提取元素，另一个栈作为队列尾部用于插入元素。当作为头部的栈为空的时候，它会与作为尾部的栈相互替换。通过 `len(q)`，`__len__` 方法能获取队列 q 中的元素数量，并通过 `if q` 测试队列是否为空。幸运的是，这些操作所需时间都是常数。

① 顺序访问所有拥有同一个下标或散列值的键，直到找到需要的对象。——译者注

```
class OurQueue:
    def __init__(self):
        self.in_stack = []            # 队列的尾部
        self.out_stack = []           # 队列的头部

    def __len__(self):
        return len(self.in_stack) + len(self.out_stack)

    def push(self, obj):
        self.in_stack.append(obj)

    def pop(self):
        if not self.out_stack:        # 队列头为空
            self.out_stack = self.in_stack[::-1]
            self.in_stack = []
        return self.out_stack.pop()
```

1.5.4　优先级队列和最小堆

优先级队列是一个抽象类型数据，能够添加元素，并取出键数字最小的那个元素。在生成哈夫曼编码（见 10.1 节）和在图中找到两个点的最短路径（见 8.3 节 Dijkstra 算法）时，利用优先级队列对一个数组进行排序（用堆排序算法），十分有用。优先级队列通常是通过堆的方式来实现的，堆的数据格式类似于一棵树。

- **满二叉树和完全二叉树**

如果一棵二叉树的所有叶子节点与根节点之间的距离都相同，则二叉树被称作**满二叉树**。如果一棵二叉树的所有叶子节点最多位于两层，所有浅层叶子节点全满，而最深层的叶子节点集中在最左边，这就是一棵**完全二叉树**。使用数组可以很容易表示这样的树形结构（图 1.3）。这棵树下标为 0 的元素被忽略，根节点的下标是 1，节点 i 的两个子节点是 $2i$ 和 $2i+1$。利用简单的计算即可操作和遍历这棵树。在第 10 章中，有其他表示树形结构的数据结构。

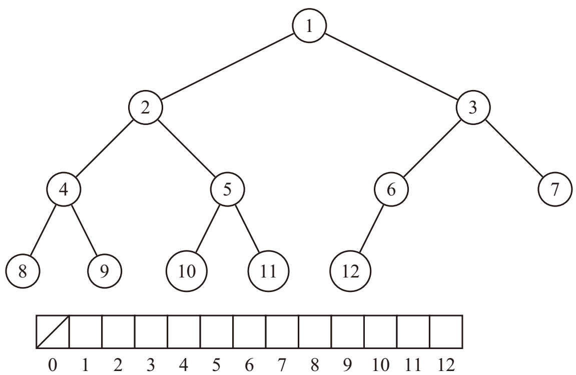


图 1.3　一棵使用数组结构表示的完全二叉树

- **优先级队列和堆**

堆（heap）是一个能检查元素优先级的反转树状结构。假如每个节点的键值（也就是优先级）比其子节点小，那这就是一个最小堆。最小堆根节点的键值一定是堆中最小的一个。同样也存在**最大堆**的概念，即每个节点的键值都比其所有子节点的键值要大。

人们通常更感兴趣的是二叉堆，即完全二叉树。这类数据结构能在**对**数时间内提取最小元素和插入新元素。总的来说，这里所讲的是有一定**顺序**关系的元素集合。堆也能更新一个元素的优先级，在使用 Dijkstra 算法**寻找**一条向顶端的最短路径时，这个操作非常有用。

在 Python 语言中，堆排列是用 `heapq` 模块实现的。这个模块提供了把数组转化成堆的方法，即 `heapify(table)`。而转化后的数组仍是前面提到的完全二叉树，唯一的区别是其根节点下标为 0 的元素非空。这个模块同样可以插入一个新元素，即 `heappush(heap, element)`，以及抽出最小元素，即 `heappop(heap)`。

相反，`heapq` 模块不能修改堆中的元素值，而这个操作在 Dijkstra 算法中可以优化时间复杂度。因此，我们推荐下面更完整的实现方式。

实现的细节

相关结构包含了 heap 数组结构，储存着一个纯粹意义上的堆；结构中还包含一个 `rank` 字典，用于查找堆中元素的下标。主要操作是 `push` 和 `pop`。当用 `push` 方法插入一个新元素时，元素被当作堆中最后一个叶子节点加入，然后，堆会根据其排序规则重新组织。使用 `pop` 方法可以提取最小元素，根节点被堆的最后一个叶子节点所替换，然后堆会再次根据自身规则重新组织。图 1.4 展示了这一过程。

操作 `__len__` 返回堆的元素数量。这个操作通过 Python 隐式地把一个堆转换成一个布尔值，比如，在堆 `h` 非空的时候，可以将 `while h` 这样的判断语句作为继续循环的条件。

堆的平均复杂度是 $O(\log n)$，但在最差情况下，由于使用了字典 `rank`，复杂度会增加到 $O(n)$。

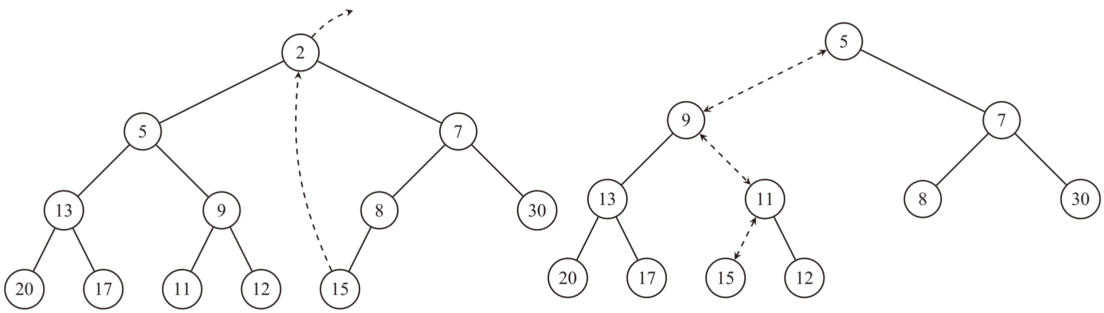


图 1.4　pop 操作移除并返回堆的数值 2，并用末端的叶子节点 15 替换。然后 down 操作执行一系列交换，将 15 移动到符合堆规则的位置[①]

① 图中是一个最小堆，其中每个节点的键值一定小于其所有子节点，因此会根据此规则执行替换。——译者注

```python
class OurHeap:
    def __init__(self, items):
        self.n = 0
        self.heap = [None]       # index 0 会被替换
        self.rank = {}
        for x in items :
            self.push(x)

    def __len__(self):
        return len(self.heap) - 1

    def push(self, x):
        assert x not in self.rank
        i = len(self.heap)
        self.heap.append(x)      # 添加一个新的叶子节点
        self.rank[x] = i
        self.up(i)               # 保持堆排序

    def pop(self):
        root = self.heap[1]
        del self.rank[root]
        x = self.heap.pop()      # 移除最后一个叶子节点
        if self:                 # 堆非空
            self.heap[1] = x     # 移动到根节点
            self.rank[x] = 1
            self.down(1)         # 保持堆排序
        return root
```

堆的重新组织通过 up(i) 和 down(i) 操作实现：当一个下标为 *i* 的元素比其父节点小，此时用 up 操作；当元素比其子节点大，则用 down 操作。因此，up 操作让某节点完成与其父节点的一系列交换，直到满足堆的规则。而 down 操作的效果类似，用于节点及其子节点的交换。

```python
    def up(self, i):
        x = self.heap[i]
        while i > 1 and x < self.heap[i // 2]:
            self.heap[i] = self.heap[i // 2]
            self.rank[self.heap[i // 2]] = i
            i //= 2
        self.heap[i] = x       # 找到了插入点
        self.rank[x] = i

    def down(self, i):
        x = self.heap[i]
        n = len(self.heap)
        while True:
            left = 2 * i       # 在二叉树中下降
            right = left + 1
            if right < n and \
               self.heap[right] < x and self.heap[right] < self.heap[left]:
                self.heap[i] = self.heap[right]
```

```
            self.rank[self.heap[right]] = i               # 提升右侧子节点
            i = right
        elif left < n and self.heap[left] < x:
            self.heap[i] = self.heap[left]
            self.rank[self.heap[left]] = i        # 提升左侧子节点
            i = left
        else:
            self.heap[i] = x      # 找到了插入点
            self.rank[x] = i
            return

def update(self, old, new):
    i = self.rank[old]            # 交换下标为 i 的元素
    del self.rank[old]
    self.heap[i] = new
    self.rank[new] = i
    if old < new:                 # 保持堆排序
        self.down(i)
    else:
        self.up(i)
```

1.5.5 并查集

- **定义**

并查集（Union-find）这种数据结构存储了一系列 V 字形集合（分片），并能完成一些指定操作。这些操作在动态数据结构中**也**被称为**查询**。

— `find(v)` 返回元素 v 所在集合内的一个特定元素。如果想检验元素 u 和元素 v 是否在同一个集合中，只需比较 `find(u)` 和 `find(v)`。

— `union(u, v)` 合并分别包含 u 和 v 的两个集合。

- **应用**

这种数据结构主要应用于检测图的元素连通性（见 6.6 节）。每次添加路径都调用一次 `union` 和 `find`，以此测试两个顶点是否在同一个集合中。并查集还可用于 Kruskal 算法对最小生成树的判断（见 10.4 节）。

- **数据结构对每个查询所需的时间基本为常量**

我们把集合中的有向树元素指向一个特定元素（图 1.5）。每个 v 元素有一个指向树中更高层级节点的引用 parent[v]。根节点 v 是集合的特定元素，在 parent[v] 中用一个特殊值来标注，我们可以选择 0 或 -1，或在值相关情况下选择 v 元素本身。整个元素的大小保存在数组 length[v] 中，其中 v 是特定元素。在这个数据结构中有两个概念。

1. 当朝向根节点遍历一个元素的时候，我们将借机压缩路径；也就是说，把遍历路径上的所有节点直接挂在根节点上。

2. 当执行合并操作 `union` 的时候，我们把序列最低的树挂在阶最高的树的根节点上。一棵树的**阶**指的是在树没有被压缩时，本应有的深度。

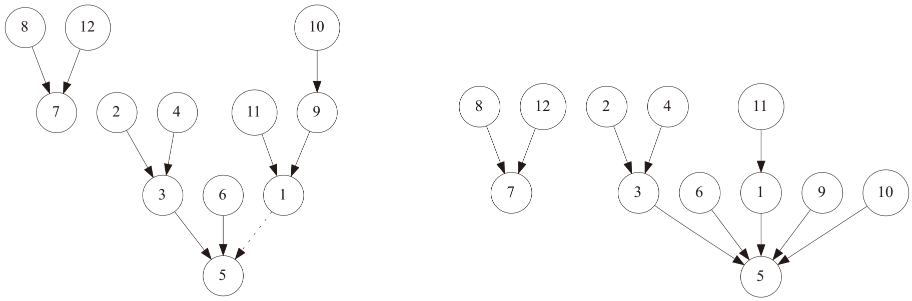


图 1.5　左图：并查集结构包含两个集合 {7, 8, 2} 和 {2, 3, 4, 5, 6, 9, 10, 11}。右图：当执行操作 `find(10)` 时，指向根节点的路径上的所有节点都直接指向根节点 5。这种机制对将来执行节点的 `find` 操作有加速作用

于是我们得到如下代码：

```
class UnionFind :
    def __init__(self, n):
        self.up = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        if self.up[x] == x:
            return x
        else:
            self.up[x] = self.find(self.up[x])
            return self.up[x]

    def union(self, x, y):
        repr_x = self.find(x)
        repr_y = self.find(y)
        if repr_x == repr_y:          # 已在同一个集合中
            return False
        if self.rank[repr_x] == self.rank[repr_y]:
            self.rank[repr_x] += 1
            self.up[repr_y] = repr_x
        elif self.rank[repr_x] > self.rank[repr_y]:
            self.up[repr_y] = repr_x
        else:
            self.up[repr_x] = repr_y
        return True
```

可以证明，对于一个大小为 n 的集合，任何 m 次 `union` 或 `find` 操作所需要的时间复杂度都是 $O((m+n)\ \alpha(n))$，其中 α 是 Ackermann 函数的反函数，一般可以视为常量 4。

1.6 技术

1.6.1 比较

在 Python 语言中，元组比较采用字典序。例如，这种方式能找到一个数组中的最大元素，同时还能找到它的下标，当有重复值的时候取最大的下标。

```
max((tab[i], i) for i in range(len(tab)))
```

举例来说，为了找到一个数组中的多数元素（majority element），我们可以用字典来统计每个元素的出现次数，并用以上代码来选择其中的多数元素。这种实现方式的平均时间复杂度是 $O(nk)$；而在最差情况下，由于使用了字典，时间复杂度是 $O(n^2k)$。其中 n 是给定输入的单词数量，而 k 是一个单词的最大长度。

这里顺便讲一下，字典数据类型的使用方式存储键值对 (key, value)。一个空字典用 `{}` 来表示。测试一个字典中是否存在键的方法是 `in` 和 `not in`。下面代码中的 `for` 循环可以遍历字典中所有的键来完成查找。

```
def majority(L):
    compute = {}
    for word in L:
        if word in compute:
            compute[word] += 1
        else:
            compute[word] = 1
    valmin, argmin = min((-compute[word], word) for word in compute)
    return argmin
```

1.6.2 排序

Python 语言中包含 n 个元素的数组排序的时间复杂度是 $O(n\log n)$。排序分为以下两种。

— `sort()` 排序：这个方法会直接修改被排序的列表内容，称为“原地”修改。

— `sorted()` 排序：这个方法会返回相关列表的一个排好序的副本。假设包含 n 个整数的数组 L，我们想在其中找到两个差值最小的整数。为了解决这个问题，可以先对数组 L 进行排序，然后对其进行遍历，最终找到数值最接近的两个整数。使用 `min` 方法结合字典排序法，可以找到集合中的多组整数对。同样，`valmin` 变量包含着数组 L 中两个连续元素的最小差值（即数组 L 中两个值最近的数的差值）；`argmin` 变量则是这两个数中较大一个数的下标。

```
def closest_values(L):
    assert len(L) >= 2
    L.sort()
    valmin, argmin = min((L[i] - L[i - 1], i) for i in range(1, len(L)))
    return L[argmin - 1], L[argmin]
```

在最差情况下，对 n 个元素排序所需的时间复杂度是 $\Omega(n\log n)$。为了证明这一点，我们假设有一个包含 n 个不同整数的数组。算法必须在 $n!$ 种可能序列中找到一种排好的序列。每次比较会返回两种可能中的一个值（更大或更小），并把结果空间切分为两部分。最终，在最坏情况下，需要 $[\log_2(n!)]$ 次比较才能找到这个特定序列，从而得到复杂度的下限 $\Omega(\log(n!)) = \Omega(n\log n)$。

- **变种**

在某些情况下，我们可以在 $O(n)$ 时间内对一个包含 n 个整数的数组进行排序。比如，一个数组内的所有整数全部在 0 到 cn 范围内，其中 c 是任意实数。我们只需遍历输入，在一个大小为 cn 的数组 count 中计算每个元素的出现次数；然后使用下标降序遍历 count，就可以得到一个包含了 0 到 cn 的值的输出数组。这种排序方法称为“计数排序”（counting sort）。

1.6.3　扫描

众多几何学问题都可以用扫描算法来解决。许多关于区间（interval），也就是一维几何对象的问题也一样。扫描算法旨在从左往右地遍历输入元素，并对每个遇到的元素做特定处理。

- **例子：区间交叉**

对于给定的 n 个区间 $[l_i, r_i)$，其中 $i = 0,\cdots, n-1$，我们希望找到一个 x 值，它被最多的区间包括。以下是一个时间复杂度为 $O(n\log n)$ 的解决方案。我们把所有极限值一起排序，然后用一个假想的指针 x 从左到右遍历这些极限值；再用一个计数器 c 来记录只看到起始值却看不到终止值的区间的数量，于是，最后这个区间数量就包含了 x。

注意，B 元素的处理顺序保证了每个区间的终止值在区间的起始值之前得到处理，这对我们处理的右侧半开放区间的情况非常必要。

```
def max_interval_intersec(S):
    B = ([(left,  +1) for left, right in S] +
          [(right, -1) for left, right in S])
    B.sort()
    c = 0
    best = (c, None)
    for x, d in B:
        c += d
        if best[0] < c:
            best = (c, x)
    return best
```

1.6.4 贪婪算法

我们在这里要介绍一种构成贪婪算法的主要算法技巧。笼统来说，这种算法在寻找解决方案的每个步骤中都选择了一个让局部结果最大化的参数。比较正式的说法是，这种算法通过拟阵组合结构，能够证明贪婪算法的优化和不优化程度。我们在本节就不对此展开讨论了（见参考文献 [21]）。

- **例子：最小点积**

我们使用一个简单的例子来介绍这种算法。对于两个给定的向量 x 和 y，它们均由 n 个正整数或空组成，首先需找到一种元素的排列 $\pi\{1,\cdots,n\}$，使得 $\sum_i x_i y_{\pi(i)}$ 最小。

- **应用**

假设以映射方式将 n 项任务交给 n 个工人完成，也就是说，每项任务必须分别分配给不同的工人。每项任务都有一个完成小时数，每个工人都有一个按每小时计算的工资数。目标是，找到一种排列方式，使得支付给工人的工资总数最少。

- **时间复杂度为 $O(n\log n)$ 的算法**

既然最佳解决方案是对 x 和 y 采用同一种排列，在不失普适性的情况下，我们可以假设 x 已经按升序排列好。假设有一个答案把 x_0 和一个最大元素 y_j 相乘，对于下标 k 且当 $y_i < y_j$ 时，有一个确定排序 π，使得 $\pi(0) = i$ 且 $\pi(k) = j$。我们会发现，$x_0y_i + x_ky_j$ 大于或等于 $x_0y_j + x_ky_i$，这意味着，在没有额外成本的情况下，π 可以变换为 x_0 乘以 y_i。证明过程如下，注意这里的 x_0 和 x_k 都是正数或为空。

$$
\begin{gathered}
x_0 \leqslant x_k \\
x_0(y_j - y_i) \leqslant x_k(y_j - y_i) \\
x_0y_j - x_0y_i \leqslant x_ky_j - x_ky_i \\
x_0y_j + x_ky_i \leqslant x_0y_i + x_ky_j.
\end{gathered}
$$

通过重复操作截断参数，从 x_0 中截断出向量 x，并从 y_j 中截断出向量 y，我们发现，当 $i \to y_{\pi(i)}$ 且 $y_{\pi(i)}$ 为逆序的时候，结果最小。

```
def min_scalar_prod(x, y):
    x = sorted(x) # 得到排好序的副本
    y = sorted(y) # 提前准备参数
    return sum(x[i] * y[-i - 1] for i in range(len(x)))
```

1.6.5 动态规划算法

动态规划算法如同程序员随身携带的瑞士军刀，是一项必备的工具。其思路是把问题分解成若干子问题，并基于子问题的解决方案找到原始问题的最优解。

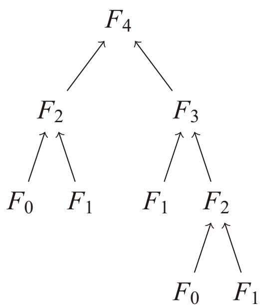
一个经典例子就是计算斐波那契数列第 n 个数的算法。斐波那契数列以如下递归方式定义：

$$F(0) = 0$$

$$F(1) = 1$$

$$F(i) = F(i-1) + F(i-2)$$

比如在我们爬楼梯的时候，这个算法可以计算在一次登上 1 或 2 级台阶的情况下，登上 n 级台阶有多少种走法。使用递归方式计算 F 效率很低，因为对于相同的参数 i，$F(i)$ 需要进行多次计算（图 1.6）。而以动态规划算法作为解决方案时，只需简单地把 $F(0)$ 到 $F(n)$ 的数值储存在一个大小为 $n+1$ 的数组中，并按照下标升序填充数组。如此一来，在计算 $F(i)$ 时，$F(i–1)$ 和 $F(i–2)$ 的值已经被计算好，并存储在数组相应的位置上。



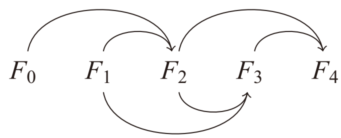


图 1.6　左边使用树状结构的穷举法实现斐波那契数列 $F(4)$ 的计算过程。右边采用动态规划算法计算依赖值[①]的方式构成了一个有向无环图，大幅减少了节点数量[②]

1.6.6　用整数编码集合

这是一种用一群集合编成整数的高效算法，集合中元素都是介于 0 至 k 的 63 次幂[③]范围内的整数。更准确地说，是使用二进制转换的方式把子集编码成特征向量。编码方式如下表所示。

---

① 即计算 $F(i)$ 时候的 $F(i–1)$ 和 $F(i–2)$。——译者注

② 左图中每个非叶子节点的值都是通过两个子节点的值计算得来，相同值的节点被多次重复计算；而右图采用动态规划算法，每个节点仅需被计算一次，减少了重复计算的次数。——译者注

③ 这个数字是来自 Python 的整数。由于 Python 的整数一般存储在一个机器字中，而这个机器字的长度如今一般是 64 个二进制位。

表 1.2

值	表达式	解释
{}	0	空集
$\{i\}$	1 << i	这个值代表2^i
$\{0,1,\cdots,n-1\}$	(1 << n) − 1	$2^n-1=2^0+2^1+\cdots+2^{n-1}$
$A\cup B$	A | B	管道运算符 | 代表二进制或
$A\cap B$	A & B	与运算符 & 代表二进制与
$(A\setminus B)\cup(B\setminus A)$	A ^ B	按位异或运算符 ^ 代表异或
$A\subseteq B$	A & B == A	测试是否包含
$i\in A$	(1 << i)& A	测试是否属于集合
$\{\min A\}$	−A & A	如果A为空，此表达式值为0

图 1.7 中给出了最后一个表达式的证明过程。这个表达式在循环计算一个集合的基数[①]时非常有用。但不存在等价算式来获取集合的最大值。

我们来看一个经典问题如何应用这一编码技巧。

获取用整数编码的集合 {3, 5, 6} 中的最小值，这个整数是 $2^3+2^5+2^6=104$。

64+32+8=104	01101000
104 的补数	10010111（每位二进制取反）
−104	10011000
−104&104 = 8	00001000（两个二进制数按位与）

结果是 2^3，从而找到单元素集 {3}

图 1.7　获取集合的最小值

- **例子：平均分三份**

定义

假设有 n 个整数 $x_0,\cdots,x_{n-1}$，现要把这些数平均分配到 3 个集合中，且每个集合中的整数和相同。

穷举方式时间复杂度为 $O(2^{2n})$ 的贪婪算法

思路是枚举所有**不相交**子集 $A, B\subseteq\{0,\cdots,n-1\}$，并比较 $f(A)$、$f(B)$、$f(C)$，其中 $C=\{0,\cdots,n-1\}\setminus A\setminus B$ 且 $f(S)=\sum_{i\in S}x_i$。这种实现方式不需要维护和比较 C 集合，只需证明 $f(A)=f(B)$ 且 $3f(A)=f(\{0,\cdots,n-1\})$。

```
def three_partition(x):
    f = [0] * (1 << len(x))
    for i in range(len(x)):
        for S in range(1 << i):
            f[S | (1 << i)] = f[S] + x[i]
    for A in range(1 << len(x)):
        for B in range(1 << len(x)):
            if A & B == 0 and f[A] == f[B] and 3 * f[A] == f[-1]:
                return(A, B, ((1 << len(x)) -1) ^ A ^ B)
    return None
```

① 即集合中包含元素的个数。——译者注

这种算法还有另一种应用：使用四则运算来计算指定值（见 15.5 节）。

1.6.7　二分查找

- **定义**

假设 f 是一个布尔函数，即值在 $\{0, 1\}$ 范围内的函数，且有如下规律：

$$f(0) \leqslant \cdots \leqslant f(n-1) = 1$$

现在要找到最小的实数 k 使得 $f(k) = 1$。

- **时间复杂度为 $O(\log n)$ 的算法**

在一个区间 $[l, h]$ 中查找，起初 $l = 0$，$h = n-1$。然后用区间的中间值 $m = [(l + h)/2]$ 来测试函数 f。根据前面的计算结果，查找空间缩小为 $[l, m]$ 或 $[m+1, h]$。注意，在计算 m 的时候向下取整，这样，第二个区间就永远不会为空，第一个区间也是。在 $[\log_2(n)]$ 次迭代后，即查找区间缩小为单元素的时候，查找会结束。

```
def discrete_binary_search(tab, lo, hi):
    while lo < hi:
        mid = lo + (hi - lo) // 2
        if tab[mid]:
            hi = mid
        else:
            lo = mid + 1
    return lo
```

- **类库**

Python 标准模块 bisect 中提供了二分查找算法，所以在某些情况下，我们不需要自己来实现。假设有一个数组 tab，由 n 个已排序好的元素组成。现在要为新元素 x 找到插入点[①]，那么需要执行 `bisect_left(tab, x, 0, n)`，而其返回值就是第一个满足 $tab[i] \geqslant x$ 的数组元素的下标 i。

- **连续域**

这种技术同样可以用在以下情况：函数 f 的区间为连续，且希望找到最小值 x_0，使得对于所有 $x \geqslant x_0$，都有 $f(x) = 1$。此时，时间复杂度取决于 x_0 需要的精确度。

```
def continuous_binary_search(f, lo, hi):
    while hi - lo > 1e-4:          # 这里设定精确度
        mid = (lo + hi) / 2.       # 浮点数除法①
        if f(mid):
            hi = mid
        else:
            lo = mid
    return lo
```

① 插入的位置要满足排序规则。——译者注

● **无上界的连续域查找**

假设 f 是一个单调布尔函数，$f(0) = 0$，且保证存在整数 n，使得 $f(n) = 1$。最小的整数 n_0 使得 $f(n_0) = 1$，即使不存在查找所需要的上限，也可以在时间 $O(\log n_0)$ 内找到 n_0 [①]。起初，我们设 $n = 1$；当 $f(n) = 0$ 时，我们把 n 翻倍。一旦找到整数 n 使得 $f(n) = 1$ 时，我们就采用通常的二分查找方法。

● **三分查找**

假设函数 f 在 $\{0,\cdots, n-1\}$ 区间内先递增，后递减，而我们要找到其中的最大值。在这种情况下，把查找区间 $[l, h]$ 拆分成三块，即 $[l, a]$、$[a+1, b]$ 和 $[b+1, h]$，这样比拆成两块更简单。通过比较 $f(a)$ 和 $f(b)$ 的值，可以判断 $[l, b]$ 和 $[a+l, h]$ 中的哪个区间包含要找的最大值。这种算法需要的迭代次数是对数 $\log_{3/2}n$。[②]

● **在区间 $[0, 2^k)$ 中的查找**

如果查找区间的大小 n 是 2 的幂，仅使用位操作中的位移运算和异或运算，就可以对普通二分查找进行少许优化。我们从数组的最后一个元素的下标开始。这个元素的二进制格式是长度为 k 的一列 1。对于每个要测试的二进制位，我们把它替换成 0，即可得到用于遍历整个数组的下标 i，而测试 tab[i] 的真伪，即可完成查找。

```
def optimized_binary_search(tab, logsize):
    hi = (1 << logsize) - 1
    intervalsize = (1 << logsize) >> 1
    while intervalsize > 0:
        if tab[hi ^ intervalsize]:
            hi ^= intervalsize
        intervalsize >>= 1
    return hi
```

● **逆函数**

对于连续且严格单调函数 f，一定存在一个逆函数 f^{-1}，后者也是单调的。假设函数 f^{-1} 的计算比函数 f 简单许多，当给定一个值 x 的时候，我们可以借它来完成对 $f(x)$ 的计算。其实，只要找到最小值 y 使得 $f^{-1}(y) \geqslant x$ 就可以了 [③]。

● **例子：填充蓄水池**

某个连通容器系统由 n 个瓶壁高度不同的容器互相连通组成，我们想计算将系统的液位提升到一个指定高度所需注入的水量。或者，假设向系统中注入体积为 V 的液体，想确定系统的液面高

---

① 单调函数又称增函数或减函数。这里的布尔函数 f 的值从 0 增加到 1，因此是增函数。如果 n_0 是使 $f(n_0)=1$ 的最小值，对于单调函数 f 一定存在一个 $n_1 > n_0$，有 $f(n_1) = 1$，且 n_0 和 n_1 间的所有值 n_x 都有 $f(n_x)=1$，这就很容易找到 n_0。——译者注

② 注意，比较 $f(a)$ 和 $f(b)$ 后迭代查找的区间不是最开始拆分开的左、中、右三个区间中的一个，而是左 + 中或中 + 右两个区间中的一个。画个图就很容易理解了。——译者注

③ 这里还是在说找单调函数里面最小最大值的问题。——译者注

度，可以使用以下方式[①]：

```
level = continuous_binary_search(lambda level: volume(level) >= V, 0, hi)
```

1.7　建议

我们在这里给出一些建议，帮助读者更快解决算法问题，并写出正确的程序。首先，要学会有组织、成体系地思考。为此，一定不要在尚未清楚理解题目的所有细节之前，仅凭一时冲动就开始编写程序。如果你在拿起键盘之前先冷静地审视一下，就不会轻易犯下某些错误，否则，你很容易写出一个根本无法实现的方案。

如果有可能，最好在竞赛时把读题和解题的时间分开。多给自己一点时间。在程序代码的注释中添加问题描述，如果有可能，再加上题目的 URL，并明确指出算法的时间复杂度。在一段时间后，当你回头再看自己编写的程序时，一定会欣赏这种做法。尤其，这能让程序代码保持逻辑严密、结构紧凑。尽量使用题目中提到的名词，以便显示答案和题目的相关性，因为没有什么比调试变量名更没有实际意义、更让人难受的事情了。

- **好好读题**

什么样的时间复杂度可以被接受？

注意题目中提出的限制，在实现你的算法之前做好复杂度分析。

输入数据是否有条件、有保证？

不要从题目的例子中猜测条件。不要做任何猜测。如果题目中没有说明“图是非空的”，那么某些测试用例中就有可能包含空的图。如果题目中没有说“字符串不包含空格”，那么就可能有一个测试用例包含这样的字符串。

使用什么样的数据类型？

整数还是浮点数？数字是否有可能是负值？如果你使用 Java 或 C++ 写程序，注意要确定中间变量的上限值，选择使用 16 位、32 位或 64 位的整数。

哪道问题更简单？

对于一个需要完成几个问题的竞赛，你应当在开始时快速浏览所有题目，分析每道题目的类型，是贪婪算法、隐式图还是动态规划算法？而后评估题目的难度。把精力集中在那些最简单且优先级最高的题目上。在团体竞赛的时候，要根据每个参赛者的专业程度来分配题目。留意其他队伍的进度，也能帮你发现容易解决的简单题目。

---

① 这里调用的 continuous_binary_search 方法是在前文“连续域查找”中定义的，可以理解为先往容器系统中倾倒液体，多了就减半再试，少了就加一半再试，直到找到符合精确度的液体量。
——译者注

- **做好计划**

比较题目的例子

画纲要图。找到待解决问题与已知问题之间的关联，如何利用实例的特殊性？

如果可能，利用类库

掌握经典的二分查找算法、排序和字典等类库。

使用题目中提到的名词命名变量

名词越简短、表述越清晰越好。

初始化变量

确保在任何新测试用例使用前，所有变量已经被重新初始化。继续上一个未完成的迭代是一个很典型的错误。举例来说，一个程序解决了图的问题，输入中包含了很多个测试用例。每个测试用例以两个整数开始：顶点数量 n 和道路数量 m。接着是两个大小都是 m 的整数组 A 和 B，其中保存了每个用例中道路的顶点。假设我们使用邻接链表来编辑一个图，对于每个 $i = 0,\cdots, m-1$，把 $B[i]$ 与 $G[A[i]]$ 相加，把 $A[i]$ 与 $G[B[i]]$ 相加。如果链表没有在每次读入测试用例前被清空，题目中的路径会累积在一起，形成一个所有图的交集。

- **调试**

现在犯错

为了以后有正确的反应[①]。

设定和测试更多的测试用例

对于有限制条件的情况（裁判回复“错误答案”）[②]和输入数据很多的情况（裁判回复“答题超时”或“运行时错误”）[③]，设定更多测试。

解释算法

解释自己的算法，并向队友评论程序。你必须能解释清楚每一行代码。

简化实现

把相似代码重新组织和重构。

冷静审视

先跳转到另一道问题上，然后回头再看，以获得新的视角。

比较

比较你的本地开发环境和要运行代码的服务器环境。

---

① 平时多调试，多看错误信息，积累定位错误的灵感。——译者注

② 此时一般没有检查边界条件。——译者注

③ 此时一般是有了死循环和除 0 错误。——译者注

1.8　走得更远

以下推荐的作品，能帮你更深入地理解本书涉及的内容。

- 基础算法:《算法导论（第 3 版）》(T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, The MIT Press: Cambridge, 2009)。
- 更特殊算法：*Encyclopedia of Algorithms* (Editors: Ming-Yang Kao, Springer Verlag, 2008)。
- 流算法相关深入、广泛的研究:《网络流：理论、算法与应用》(R.K. Ahuja, T.L. Magnanti and J.B. Orlin, Prentice Hall, 2011)。
- 几何算法：*Computational Geometry: Algorithms and Application*s (M. de Berg, O. Cheung, M. van Krevel and M. Overmars, Springer Verlag, 2011)。
- 其他广受欢迎的参考书：*Python Essential Reference* (David M. Beazley, Pearson Education, 2009) 和《Python cookbook，第三版》(David M. Beazley, Brian K. Jones，人民邮电出版社，2015 年)。
- 对于准备竞赛很有帮助的书：*Competitive Programming* (Steven and Felix Halim, Lulu, 2013)。
- 最后是《算法设计指南》(Steven S. Skiena, Springer Verlag, 2009 年)。

本书最后给出了书中提及的参考文献，其中包含了图书和科研论文。但对于大众来说，论文并不容易接触到。读者可以通过 Google Scholar 或在大学图书馆里尝试寻找这些文献的原文。

阅读本书有时需要配合使用网站 tryalog.org/index-en。在这里，读者不但能找到本书中编写的 Python 程序，还能找到测试用例的文件。当然，这些程序和文件也可以在 Github 和 PyPI 仓库中找到，在 Python 3 环境下使用命令 pip install tryalgo 就可以一步安装。

第 2 章　字符串

字符串处理是算法领域里非常重要的内容，其中有些是关于文字处理的，比如语法检查，有些则关于子字符串（子串）；或者更笼统地说，是关于模式（pattern）查找的。随着生物信息学的发展，出现了 DNA 序列问题。本章中将介绍一系列我们认为的重要算法。

在计算机系统内，一个字符串可以用一个字符的列表来表示。但一般情况下，我们会使用 `str` 类型，这个类型在 Python 语言中类似于一个列表。对于使用 Unicode 编码方式的字符串，每个字符可以使用两个字节来编码。一般情况下，字符仅用一个字节表示，并用 ASCII 码来编码：0 ～ 127 的每个整数代表一个不同的字符，编码按顺序排列，如 0 ～ 9、a ～ z、A ～ Z。同样，如果一个字符串只包含大写字母，我们可以使用 ord($s[i]$) – ord('A') 这样的计算方式找到第 i 个字符在字母表中的位置。反过来说，第 j 个（从 0 开始编号）**大写字母**可以使用 chr(j+ord('A')) 找到。

说到子串，也就是字符串的子字符串的时候，一般要求字符必须是连续的[①]，这与更通常的子序列（子字）的定义不同。

2.1　易位构词

- **定义**

如果对调字符，使得单词 w 变成单词 v，那么 w 就是 v 的易位构词。假设有一个集合包含了 n 个最大长度为 k 的单词，现在要找到所有的易位构词。

输入：le chien marche vers sa niche et trouve une limace de chine nue pleine de malice qui lui fait du charme

输出：{une nue}, {marche charme}, {chien chine niche}, {malice limace}.

---

① 即中间没有空格。——译者注

句子的意思是："一条狗走向狗窝时遇到一条顽皮的鼻涕虫，被吸引了过去。"其中某些单词，如 chien（狗）和 niche（窝）、limace（鼻涕虫）和 malice（顽皮）等，都是字母相同而顺序不同的单词，输出得到的是输入句子中所有易位构词的集合。

- **复杂度**

以下算法能在平均时间 $O(nk\log k)$ 内解决问题。而在最坏情况下，由于使用了字典，所需时间复杂度是 $O(n^2k\log k)$。

- **算法**

算法的思路是计算每个单词的签名。两个单词能得到相同的签名，当且仅当它们互为易位构词。这个签名不过是包含了相同字母的另一个单词，是把要计算签名的单词中的所有字母按顺序排列后得到的。

算法使用的数据结构是一个字典，将每个签名与拥有这一签名的所有单词的列表对应起来。

```
def anagrams(w):
    w = list(set(w))                    # 删除重复项
    d = {}                              # 保存有同样签名的单词
    for i in range(len(w)):
        s = ''.join(sorted(w[i]))       # 签名
        if s in d:
            d[s].append(i)
        else:
            d[s] = [i]
    # -- 提取易位构词
    reponse = []
    for s in d:
        if len(d[s]) > 1:               # 忽略没有易位构词的词
            reponse.append([w[i] for i in d[s]])
    return reponse
```

2.2 T9：9 个按键上的文字

输入：2 6 6 5 6 8 7

输出：bonjour

- **应用**

按键式移动电话提供了一种有趣的输入方法，通常被称作 T9 输入法。26 个字母分布在数字 2 ～ 9 的按键上，就像图 2.1 展示的一样。为了输入一个单词，只需按对应的数字键就可以了。但是，有时输入一个相同的数字序列却可能得到不同的单词。在这种情况下，就需要用字典来推测最有可能出现的单词，并把这些单词摆放在候选词的首位。

- **定义**

这个问题实例的第一部分是一个字典结构，由一系列键值对 (m, w) 组成，其中 m 是一个由 26 个小写字母中部分字母组成的单词，w 是这个单词的权重。问题实例的第二部分由输入序列为 2 ～ 9 的数字组成。对于每个输入序列，只需要显示字典中权重最高的一个单词。假设有一个数字序列使用 T9 输入法，根据图 2.1 中的对应关系，输出单词为 m，而输入数字序列 t 是通过将单词 m 中的每个字母都替换为相关数字得来的。s 是输入数字序列 t 的前缀，这时，我们就可以定义单词 m 与 s 相关。比如单词 bonjour（你好）与数字序列 26 相关，也和数字序列 266 或 2665687 相关。



图 2.1　一个移动电话键盘上的按键

- **复杂度为 $O(nk)$ 的算法**

字典初始化的时间复杂度为 $O(nk)$，而每次查询的时间复杂度为 $O(k)$。这里的 n 是字典中的单词数量，k 是单词长度的上限。

在第一时间，对于字典中某个单词的每个前缀 p[①]，我们要查找将 p 作为前缀的所有单词的总权重，并把总权重存入一个 freq（频率）字典中。接下来，我们在另一个字典 prop[seq] 中存储赋予每个给定的 seq 序列的前缀列表。遍历 freq 中的所有键，可以确定权重最大的前缀。此处的关键就是 word_code 函数，它能为给定单词提供相关的输入数字序列。

为方便阅读，以下算法实现的时间复杂度是 $O(nk^2)$。

```
t9 = "22233344455566677778889999"
#      分别对应 abcdefghijklmnopqrstuvwxyz 这 26 个字母

def letter_digit(x):
    assert 'a' <= x and x <= 'z'
    return t9[ord(x)-ord('a')]

def word_code(words):
    return ''.join(map(letter_digit,words))

def predictive_text(dico):     # dico 意为字典
    freq = {}                  # freq[p] = 拥有前缀 p 的单词的总权重
```

① 这里可以理解为，每次用 T9 输入法输入一个新数字的时候，由于尚未输入完成，输入数字序列的前几个数字就是整个输入序列的前缀。——译者注

```
    for words,weights in dico:
        prefix = ""
        for x in words:
            prefix += x
            if prefix in freq:
                freq[prefix] += weights
            else:
                freq[prefix] = weights
    #   prop[s] = 输入 s 时要显示的前缀
    prop = {}
    for prefix in freq:
        code = word_code(prefix)
        if code not in prop or freq[prop[code]] < freq[prefix]:
            prop[code] = prefix
    return prop

def propose(prop, seq):
    if seq in prop:
        return prop[seq]
    else:
        return "None"
```

2.3　使用字典树进行拼写纠正

- **应用**

如何把单词存入一个字典来纠正拼写呢？对于某个给定的单词，我们希望很快在字典中找到一个最接近的词。如果把字典里的所有单词存在一个散列表里，单词之间的一切相近性信息都将丢失。所以，更好的方式是把这些单词存入字典树，字典树也叫前缀树或排序树（trie tree）。

- **定义**

一棵保存了某个单词集合的树称为字典树。连接一个节点及其子节点的弧线用不同字母标注。因此，字典中的每个单词与树中从根节点到树节点的路径相关。每个节点都是标记，用于区分相关字母组合究竟是字典中的单词，还是字典中单词的前缀（图 2.2）。

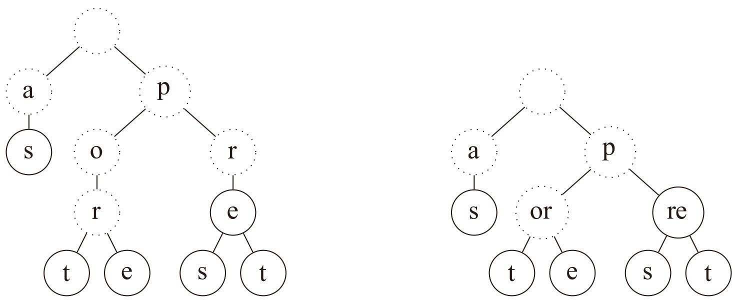


图 2.2　字典树

字典树存储着法语单词 as、port、pore、pré、près 和 prêt（但没有重音符号）。图中的虚线圈表示子节点[①]；实线圈代表字典中一个完整的单词。右边是一个前缀树代表的相同字典[②]。

- **拼写纠正**

利用上述数据结构，我们很容易在字典中找到一个与给定单词距离为 `dist` 的单词。这里的距离以**编辑距离**（levenshtein distance）来定义，本书 3.2 节有详细介绍。查找方式是只需模拟每个节点的拼写操作，然后使用参数 `dist-1` 进行递归调用。

- **变种**

若某个节点只有一个子节点，就可以合并多个节点，这种结构更精简。这种节点用单词标记，而不是用字母标记。图 2.2 右侧的结构更节省内存和遍历时间，被称为前缀树（patricia trie）。

```
from string import ascii_letters              # 在 python2 中需要引用 letters 库

class Trie_Node:
    def __init__(self):
        self.isWord = False
        self.s = {c: None for c in ascii_letters}

def add(T, w, i=0):
    if T is None:
        T = Trie_Node()
    if i == len(w):
        T.isWord = True
    else:
        T.s[w[i]] = add(T.s[w[i]], w, i + 1)
    return T

def Trie(S):
    T = None
    for w in S:
        T = add(T, w)
    return T

def spell_check(T, w):
    assert T is not None
    dist = 0
    while True:                               # 尝试用越来越长的距离来查找
        u = search(T, dist, w)
        if u is not None:
            return u
        dist += 1

def search(T, dist, w, i=0):
    if i == len(w):
```

① 这个路径的字母组合只是一个正确拼写的单词前缀。——译者注

② 右侧的树合并了只有一个子节点的路径，经过优化的结构更简洁，效率更高。——译者注

```
        if T is not None and T.isWord and dist == 0:
            return ""
        else:
            return None
    if T is None:
        return None
    f = search(T.s[w[i]], dist, w, i + 1)            # 相关
    if f is not None:
        return w[i] + f
    if dist == 0:
        return None
    for c in ascii_letters:
        f = search(T.s[c], dist - 1, w, i)           # 插入
        if f is not None:
            return c + f
        f = search(T.s[c], dist - 1, w, i + 1)       # 替换
        if f is not None:
            return c + f
    return search(T, dist - 1, w, i + 1)             # 删除
```

2.4　KMP（Knuth-Morris-Pratt）模式匹配算法

输入：lalopalalali lala

输出：　　　^

- **定义**

给定一个长度为 n 的字符串 s 和一个长度为 m 的待匹配模式字符串 t，我们希望找到 t 在 s 中第一次出现时的下标 i。当 t 不是 s 的子串时，返回值应该是 -1。

复杂度：$O(n+m)$，见参考文献 [19]。

- **穷举算法**

这种算法用来测试所有 t 在 s 中可能出现的位置，并逐个比较字符，检查 t 是否与 $s[i,\cdots,i+m-1]$ 相关。最坏情况下的时间复杂度是 $O(nm)$。下面演示了使用穷举算法的对比查找过程。每一行对应选择的一个 i，并用字母标识出在选定 i 时的相关字符，若字符不相关就用 × 来标记。

	l	a	l	o	p	a	l	a	l	a	l	i
0	l	a	l	×								
1		×										
2			l	×								
3				×								
4					×							
5						×						
6							l	a	l	a		

在处理 i 后，我们能了解字符串 s 的大部分内容。利用这些信息，就不必对例子中的 $t[0]$ 和 $s[1]$ 进行比较了。

- **算法**

我们把两个字符串 x 和 y 的重叠部分称为最长单词，这个最长单词既是 y 的严格后缀，又是 x 的严格前缀。在发现 $s[i]$ 和 $t[j]$ 有差异的时候，我们把 t 向 s 的尾部移动（从 0 到 $i-1$），以便进行后续比较。由于 $s[0,\cdots,1]$ 的前缀是 $t[0,\cdots,j-1]$（最后 j 次比较已经证明了 $s[i-j,\cdots,i-1]$ 和 $t[0,\cdots,j-1]$ 相等），因此 t 向后移动的距离仅由 t 来决定。

我们可以通过预先计算来确定 t 向后偏移的距离。用 $r[j]$ 来记录 j 减去自身与 $t[0,\cdots,j-1]$ 的重叠部分的差值。下面的程序展示了具体实现方式。为了分析复杂度，我们把计算 r 的代码和字符串匹配的代码分开：第一部分代码的复杂度是 $\Theta(m)$，第二部分代码的复杂度是 $\Theta(n)$。每当 $s[i]=t[j]$ 时，都需要把 j 增加 1；而每次两者不相等的时候，要把 j 减少 1，因为 $r[j]<j$。既然 $s[i]$ 和 $t[j]$ 最多只有 n 次相等，而且 j 永远是非负值，那么两者不相等的次数最多也只有 n。①

```
def knuth_morris_pratt(s, t):
    assert t != ''
    len_s = len(s)
    len_t = len(t)
    r = [0] * len_t
    j = r[0] = -1
    for i in range(1, len_t):
        while j >= 0 and t[i - 1] != t[j]:
            j = r[j]
            j += 1
            r[i] = j
    j = 0
    for i in range(len_s):
        while j >= 0 and s[i] != t[j]:
            j = r[j]
        j += 1
        if j == len_t:
            return i - len_t + 1
    return -1
```

① 第一步预处理计算了带匹配的模式字符串 t 的每个子字符串的最大前缀和后缀的公共元素长度，即 t 本身包含的重复字符和字符组合。这样一来，每次匹配失败时，带匹配字符串不是通过简单地向后移动一位来继续查找，而是根据预先算好的前缀和后缀的公共元素表来跳过一定数量的字符，以此直接匹配 t 中重复的字符串或字母组合，从而提高效率。比如，字符串 t 是 ABCAD，字符串 s 是 DE**ABCAB**ABCADE。t 中两个 A 重复出现，第一次 ABCAD 匹配 ABCAB 在最后一个字符 D 和 B 比较时失败，此时，我们准确地知道匹配失败的字符 D 的前一个字符 A 匹配成功了，即 ABCA 都匹配成功了，那么我们就不再需要比较 s 中的其他字母。也就是说，不是将 t 中的 A 和 s 中的 B 比较，而是直接用已经匹配成功的 t 中的 A 来和 s 中的 A 对齐。再次强调，由于 t 中有两个 A 重复，而其他字符都不是 A，那么我们希望匹配 s 中的 A 时，只能用 t 中的两个 A 中的一个来对齐 s 中的 A，这样就跳过了一定不相等的 B 和 C 等字符。——译者注

- **变种**

在不增加复杂度的情况下，增加一个很小的改动可以生成一个大小为 n 的布尔型数组 p，它指明了对于每个位置 i，t 是否是 s 在 i 位置的一个子串。概括地说，我们可以计算出一个整数数组 p，它判断了对于每个位置 i 是否有最大的 j，使得长度为 j 的 t 的最大前缀字符串是 s 在 i 结尾的子字符串。这个算法将在后面介绍。

2.5　最大边的 KMP 算法

查找一个字符串的最大边，也可以帮助我们解决字符串的模式匹配问题。这一算法的基本思想与 KMP 模式匹配算法相同，但使用了更多技巧，因此实现方式也更简洁。

- **定义**

当字符串 w 的某个子字符串同时是 w 本身的严格前缀和严格后缀时，我们把该子字符串称作字符串 w 的边，且将最大边记为 $\beta(w)$。举例来说，字符串 abababa 的边有 aba、a 和空字符串 ε。对于一个给定字符串 $w = w_0,\cdots,w_{n-1}$，现在要计算 w 的每个前缀的最大边，也就是计算这些边的长度，因为 w 的前缀的边同时也是 w 的前缀。因此，我们也可以快速找到前缀长度的序列 $l_i = |\beta(w_0,\cdots,w_{n-1})|$。

- **关键测试**

按照边的思路来观察一个递归结构：假设 u 是 v 的边，而 v 是 w 的边，那么 u 同时也是 w 的边。用 β 对一个字符串 w 进行迭代运算，就能得到 w 所有的边。比如，对于 w = abaababa，可以得到 $\beta(w) = aba$，$\beta(\beta(w)) = a$，以及 $\beta^3(w) = \varepsilon$。

- **算法**

假设已知字符串 w 的前 i 个前缀的最大边，即已知前缀 $u = w_0,\cdots,w_{i-1}$（也就是说，子字符串 $w_0,\cdots, w_{i-1}$ 拼接而成的字符串 u）。让我们先考虑前缀 ux（更明确地说，x 表示字符 w_i）：其最大边的形式一定是 vx，其中 v 是 u 的边（图 2.3）。我们用 $v_j = \beta^j(u)$ 来记录字符串 u 按长度降序排列的第 j 条边，并用 k_j 来记录这条边的长度。为此，要从最大边 $v_1 = \beta(u)$ 开始，在 u 的边的序列 (v_j) 中寻找 v。为了检测一个已经是 ux 的后缀的候选边 v_jx 是否是 ux 的边，只需确认 v_jx 是否是 ux 的前缀即可。然而，既然 v_j 已经是 u 的前缀（即一条边），那么只需要测试紧接着 v_j 后续（也就是在位置 $k_{j} = |v_j|$）的字符是否是 x。这就又回到了测试 $W_{kj} = ?x$ 。如果满足条件，那么就找到了 $\beta(ux)= v_jx$，对 ux 的最大边计算也就完成了。否则，就要测试下一条 $v_{j+1} = \beta(v_j)$，其长度为 $k_{j+1} = |\beta(v_j)| = |\beta(w_0...w_{kj-1})| = \ell_{kj}$，因为 v_j 恰恰是 w 的前缀且长度为 k_j。如果任何比较都没有得到想要的结果，就可以确认 $\beta(ux) = \varepsilon$。因为在每次迭代中，我们进行的唯一一次测试，也就是计算 k_{j+1}，只依赖于当前 k_j 的边长。算法的实现只需用一个变量 k 来确定数组 l。

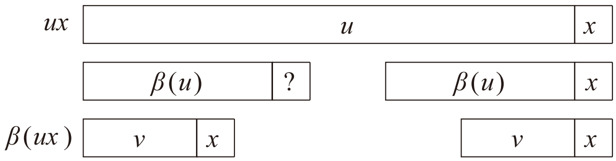


图 2.3 KMP 字符串匹配算法变种的一个计算步骤

一旦已知 u 的所有边，我们就知道 ux 的最大边形式一定是 vx，其中 v 是 u 的边。如果图中问号代替的字符是 x，那么 $v = \beta(u)$，否则就要在 $\beta(u)$ 更短的边中查找 v。

- **复杂度**

有趣的是，这个算法的复杂度呈线性下降：实际上，`while` 循环迭代的次数永远都不会超过当前边长度 k，而每次 k 在 `for` 循环中最多只增加 1。

```
def maximum_border_length(w):
    n = len(w)
    L = [0] * (n + 1)
    for i in range(1, n):
        k = L[i]
        while w[k] != w[i] and k > 0:
            k = L[k]
        if w[k] == w[i]:
            L[i + 1] = k + 1
        else:
            L[i + 1] = 0
    return L
```

- **变种**

借助最大边的列表，我们能解决很多与字符串和单词相关的问题：计算平方子串；确定回文前缀；判断两个单词 x 和 y 是否共轭，也就是说，格式是否满足对于单词 u 和 v 有 $x = uv$ 且 $y = vu$；检测一个单词 x 的最小周期 ①，即单词 x 和 z 有最大的 k 值，令 $z^k = x$。

- **关键测试**

如果字符串 u 呈周期性，则 u 的最大边是 z^{k-1}，其中 k 是当存在一个字符 z 并使得 $u = z^k$ 成立时的最大整数（图 2.4）。

```
def powerstring_by_border(u):
    L = maximum_border_length(u)
    n = len(u)
    if n % (n - L[-1]) == 0:
        return n // (n - L[-1])
    return 1
```

① 最小子字符串重复的最多次数即为最小周期。——译者注

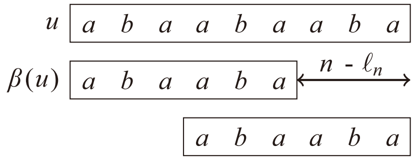


图 2.4　已知一个周期性字符串 u 的最大边，就能找到其最小周期。假设 n 是字符串 u 的长度，如果 n-l_i 能把 n 整除，那么该字符串就是周期性的。而且，若对于字符 z 有 $u = z^k$，那么其中 k 的最大值是 $n/(n$-$l_n)$

- **应用：在 s 中匹配模式字符串 t**

我们选择一个字符 #，它既不在 s 中也不在 t 中。我们关注的是字符串 $t\#s$ 的前缀最长边的长度：首先要注意的是，因为存在字符 #，这一长度绝对不会超过 t 的长度。但是，每当该长度达到 $|t|$ 时，说明我们在 s 中找到了一次 t 的存在。因此，我们可以使用动态规划算法确定字符串 $t\#s$ 的所有带有前缀 u 的列表 $l_i = |\beta(u)|$（图 2.5）。

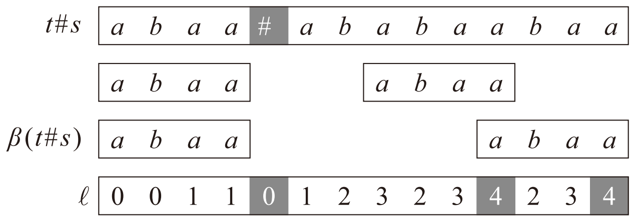


图 2.5　查找最大边算法的一次迭代。t 在 s 中每出现一次，都对应着在最大边 l 的长度列表中一条长度为 $|t|$ 的边

- **备注**

所有编程语言的标准类库都会提供一个在字符串 `haystack` 中查找模式 `needle` 的方法[①]。在 Java 8 中，该方法在最坏情况下的时间复杂度是 $\Theta(nm)$，效率低得惊人。读者可以测试一下，用这个方法计算当 n 变化时，在字符串 0^{2n} 中查找 0^n1 所需的时间[②]。

① 正如在草堆中查找一根针。——译者注

② 在 $2n$ 个 0 构成的字符串中查找 n 个 0 拼接一个 1 的字符串，这就是前面所说的最坏情况。作者提出这个问题是为了提示读者，在竞赛时使用 Java 8 的方法可能会降低效率。——译者注

2.6 字符串的幂

输入	输出
abcd	$=(\text{abcd})^1$
aaaa	$=\text{a}^4$
ababab	$=(\text{ab})^3$

- **应用**

假设你获得一个周期信号的取样结果，需要找到该信号的最短周期。此问题可以简化为确定一个最短周期，使得输入内容总是这一最短周期的多次重复。

- **定义**

设一个字符串 x，找到一个最大整数 k，使得存在一个字符串 y，令 $x = y^k$。这里 y 的 k 次幂被定义为字符串 y 拼接 k 次。问题至少有一个结果，因为 $x = x^1$。

解 k 除以长度 m 等于 x，同时对于 $p = m/k$，x 中每个字符都应该与下标较远的字符 p 相等，此时 x 被视为**圆周字符串**。在圆周字符串 x 中，最后一个字符串之后的字符被定义为字符串 x 的第一个字符。转动一次字符串 x 会把其第一个字符删掉，并将该字符添加到字符串尾部。当转动操作的执行次数等于字符串 x 中的字符个数时，字符串 x 变换后仍是字符串 x。

- **线性时间复杂度的算法**

问题变为寻找最小的 p（$p \geqslant 1$），使得字符串 x 在进行 p 次转动后仍等于 x。这里要使用圆周字符串算法中的一个经典技巧：在字符串 xx（x 后接 x）中查找 x 第一次出现的位置——当然，要去掉第 0 个位置（图 2.6）。

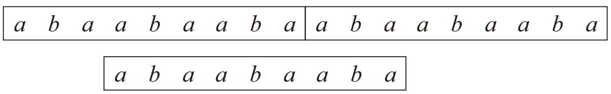


图 2.6　如果字符串 x 在 4 次转动后仍得到 x，那么字符串 x 的最小周期是 4

```
def powerstring(x):
    return len(x) // (x + x).find(x, 1)
```

2.7 模式匹配算法：Rabin-Karp 算法

复杂度：一般为 $O(n+m)$，最差情况为 $O(nm)$。

- **算法**

Rabin-Karp 算法（见参考文献 [17]）与 KMP 模式匹配算法基于完全不同的思路。为了在大字

符串 s 中找到字符串 t，应该在 s 上滑动一个长度为 len(t) 的窗口，然后判断这个窗口的内容是否与 t 相等。逐个对比字符串所需的时间成本太高，所以需要计算当前窗口内容的散列值。比较窗口内容和字符串 t 的散列值，速度会更快。当两个字符串的散列值吻合时，再进行耗时较长的逐个比较字符串操作（图 2.7）。因此，为了得到更好的时间复杂度，我们需要一个高效的方法来获取到当前窗口内容的散列值，这就要用到滑动散列函数（hash function）。

如果散列函数值范围为 $\{0, 1,\cdots, p-1\}$，而且选择得当，那么"发生碰撞"的概率能达到 $1/p$。所谓碰撞，指的是两个长度相等、**顺序**一致的独立字符串 s 和 t 被提取出同样的散列值。在这种情况下，当前算法的平均时间复杂度是 $O(n+m+m/p)$。算法实现方式使用的是 $p = 2^{31}-1$，因此在实际情况下，算法时间复杂度是 $O(n+m)$①，而最坏情况下的算法时间复杂度是 $O(nm)$。

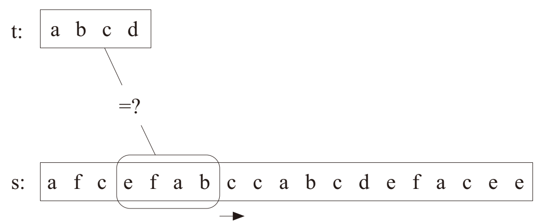


图 2.7　Rabin-Karp 算法的思路是首先比较 t 和 s 中窗口的散列值，然后再逐个比较字符串

- **计算滑动窗口散列值的方法**

散列函数首先把包含 m 个字符的字符串变换成包含 m 个整数的序列 $x_0,\cdots, x_{m-1}$，并与字符的 ASCII 码对应，整数介于 0 至 127 之间②。因此，散列函数值有如下多线性表示法：

$$h(x_0,\cdots, x_{m-1}) = x_0 \cdot 128^{m-1}+ x_1 \cdot 128^{m-2}+,\cdots, +x_{m-2} \cdot 128+ x_{m-1} \bmod p$$

其中所有操作都被一个大质数 p 进行了取模运算（modulo）。在实际操作中要特别注意，所有计算值都应能被一个 64 位机调用，其中一个机器字（处理器的寄存器）应当处在 -2^{63} 到 $2^{63}-1$ 之间。最大的中间临时变量是 $128 \cdot (p-1)=2^7 \cdot (p-1)$，这也是算法实现中选择 $p < 2^{56}$ 的原因。

这一散列函数的多项式形式可在常数时间内通过 x_0、x_m 和 $h(x_0,\cdots, x_{m-1})$ 计算 $h(x_1,\cdots, x_m)$ 值：抽取第一个字符等价于抽取多项式的第一项，字符串左移等价于多项式乘以 128，修改最后一个字符等价于添加多项式的一项。于是，让窗口在字符串 s 上移动，更新窗口内字符串的散列值并与字符串 t 进行比较，都可以在常数时间内完成。注意，把字符串向右移动等价于字符串乘以 128 对 p 取模的倒数，其运算时间也是常数③。

① 因为 p 值很大，使得 m/p 值小到可以忽略。——译者注

② 128 的乘方可以用二进制位移运算，不会耗费太多时间。——译者注

③ 假设从左向右移动窗口，那么每次移动窗口都要移除字符串 t 最左边的字符，并在最右边添加字符。用多项式表示该操作，等同于添加多项式的项，并将全部项乘以 128。——译者注

在以下代码中，在散列值中加 DOMAIN*PRIME 是为了保证计算结果是正值或空值。这在 Python 语言中并不是必要的，但在 C++ 等其他语言中，取模运算可能会得到负的返回值。

```
PRIME = 72057594037927931               # < 2^{56}
DOMAIN = 128

def roll_hash(old_val, out_digit, in_digit, last_pos):
    val = (old_val - out_digit * last_pos + DOMAIN * PRIME) % PRIME
    val = (val * DOMAIN) % PRIME
    return(val + in_digit) % PRIME
```

算法的实现从逐个比较长度为 k 的子串开始，即从在字符串 s 中位于 i 的字符和在字符串 t 中位于 j 的字符开始，比较后面的 k 个字符。

```
def matches(s, t, i, j, k):
    for d in range(k):
        if s[i + d] != t[j + d]:
            return False
    return True
```

接下来实现真正意义上的 Rabin-Karp 算法，首先计算 t 的散列值和 s 第一个窗口中字符串的散列值，然后将 s 中的所有子字符串循环。

```
def rabin_karp_matching(s, t):
    hash_s = 0
    hash_t = 0
    len_s = len(s)
    len_t = len(t)
    last_pos = pow(DOMAIN, len_t - 1) % PRIME
    if len_s < len_t :
        return -1
    for i in range(len_t):              # 预计算
        hash_s = (DOMAIN * hash_s + ord(s[i])) % PRIME
        hash_t = (DOMAIN * hash_t + ord(t[i])) % PRIME
    for i in range(len_s - len_t + 1):
        if hash_s == hash_t :           # 逐个比较字符
            if matches(s, t, i, 0, len_t):
                return i
        if i < len_s - len_t:
            hash_s = roll_hash(hash_s, ord(s[i]), ord(s[i+len_t]), last_pos)
    return -1
```

Rabin-Karp 算法比 KMP 模式匹配算法的效率略低，根据我们的测试结果，前者的运算时间是后者的 3 倍。但 Rabin-Karp 算法的优势在于，能在多个变种问题中应用自如。

变种 1：匹配多个模式

利用 Rabin-Karp 算法，在给定字符串 s 中查找 t 的问题可以拓展为在字符串 s 中查找一个字符串集合 τ 的问题，其中 τ 的所有字符串长度必须一致。为解决问题，仅需把 τ 中所有字符串的散列值存入字典 `to_search`，然后检测 s 每个窗口的散列值能否在字典 `to_search` 中找到相关值。

变种 2：公共子串

给定字符串 s、t 和一个长度值 k，寻找一个长度为 k 的字符串 f，令 f 同时是 s 和 t 的子字符串。为解决问题，首先考虑字符串 t 中长度为 k 的所有子串。这些子串都可以通过与 Rabin-Karp 算法类似的方式获得，即在 t 上滑动宽度为 k 的窗口，把获得的散列值存入字典 `pos`。每次获得一个散列值的时候，将其与窗口位置关联。

然后，对于在字符串 s 中每个长度为 k 的子串 x，检查其散列值 v 是否存在于字典 `pos` 中，如果存在，再将 x 与 t 中位于 pos[v] 位置的所有子串逐一比较。

使用这一算法时，需要选择恰当的散列函数。如果 s 和 t 的长度都是 n，为了让字符串 s 和 t 各自 $O(n)$ 个窗口中的一个窗口相互碰撞次数为常数，需要选择函数 $p \in \Omega(n^2)$①。读者可以参看参考文献 [17] 来获得更细致的解答。

变种 3：最长公共子串

给定两个字符串 s 和 t，寻找其最长公共子串，这个问题也可以采用上述算法，并以二分查找最长距离 k 的思路来解决。算法的时间复杂度是 $O(n\log m)$，其中 n 是 s 和 t 的总长度，而 m 是优化子串的长度。

```python
def rabin_karp_factor(s, t, k):
    last_pos = pow(DOMAIN, k - 1) % PRIME
    pos = {}
    assert k > 0
    if len(s) < k or len(t) < k:
        return None
    hash_t = 0
    for j in range(k):                # 存入散列值列表
        hash_t = (DOMAIN * hash_t + ord(t[j])) % PRIME
    for j in range(len(t) - k + 1):
        if hash_t in pos:
            pos[hash_t].append(j)
        else:
            pos[hash_t] = [j]
        if j < len(t) - k:
            hash_t = roll_hash(hash_t, ord(t[j]), ord(t[j + k]), last_pos )
    hash_s = 0
    for i in range(k):                # 预计算
        hash_s = (DOMAIN * hash_s + ord(s[i])) % PRIME
    for i in range(len(s) - k + 1):
        if hash_s in pos:             # 此散列值是否存在于 s 中?
            for j in pos[hash_s]:
                if matches(s, t, i, j, k):
                    return (i, j)
        if i < len(s) - k:
            hash_s = roll_hash(hash_s, ord(s[i]), ord(s[i + k]), last_pos)
    return None
```

① 碰撞次数过多会影响散列算法的性能，所以需要更大范围的值。——译者注

2.8 字符串的最长回文子串：Manacher 算法

输入：babcbabcbaccba

输出： abcbabcba

- **定义**

如果字符串 s 的第一个字符等于最后一个字符，而第二个字符又等于倒数第二个字符，**以此类推**，那么该字符串就是一个回文字符串。“最长回文子串问题”就是要找到一个最长子串，同时该子串是一个回文子串。

- **复杂度**

采用贪婪算法需要二次方的时间复杂度；采用后缀表算法需要的时间复杂度是 $O(n\log n)$；采用 Manacher 算法（见参考文献 [23]）需要的时间复杂度是 $O(n)$。

- **算法**

首先在输入字符串 s 的每个字符前后都添加 # 作为分隔符，在整个字符串的首尾添加 ^ 和 $ 字符，比如，abc 会被变换成 ^#a#b#c#$。变换后的字符串 s 用 t 来记录。这样做的好处是能够用相同方法找到长度为奇数和偶数的回文子串。注意，在使用这种转换方式时，所有回文子串都以分隔符 # 起始和结束。因此，每个回文子串的边界字符下标就拥有相同偶性①，这样一来就很容易能将字符串 t 的解决方法转换为字符串 s 的解决方法。分隔符的存在方便了字符串边界字符的处理。

> **单词 nonne 包含一个长度为 2 的回文串 nn 和一个长度为 3 的回文串 non。在转换后，字符串都用分隔符 # 来开始和结束：**
>
> ```
> | ----- |
> ^#n#o#n#n#e#$
>      | --- |
> ```

算法的输出是一个数组 p，它能指出对于每个位置 i，是否存在某个最长半径 r，使得从 $i-r$ 到 $i+r$ 位置的子串是一个回文子串。贪婪算法如下：对于所有 i，初始化 $p[i]=0$，然后递增 $p[i]$ 直至找到以 i 为中心的最长回文子串 $t[i-p[i],\cdots,i+p[i]]$。

想要优化 Manacher 算法就要初始化 $p[i]$。已知一个以 c 为中心、r 为半径的回文子串，也就是说，子串的结尾是 $d = c+r$。而 j 是 i 相对于 c 的对称镜像（图 2.8）。$p[i]$ 和 $p[j]$ 之间有着很强的关联。在 $i+p[j]$ 不超过 d 的情况下，我们可以用 $p[j]$ 来初始化 $p[i]$。这一操作十分有效：假如以 j 为中心、$p[j]$ 为半径的回文子串包含在以 c 为中心、$d-c$ 为半径的回文子串的前一半中，那么它一定也存在于后一半中。

在成功初始化 $p[i]$ 后，需要更新 c 和 d，以保存用 $d-c$ 最大值编写的回文子串的不变量。算法

---

① 对于长度为奇数的回文串，如 aba，转换后 ^#a#b#a#$ 的边界字符 a 的下标 2 和 6 都是偶数，对于长度为偶数的回文串，如 abba，转换后 ^#a#b#b#a#$ 的边界字符 a 下标 2 和 8 也都是偶数。——译者注

的时间复杂度呈线性，因为每次比较字符都会导致 d 的增加。

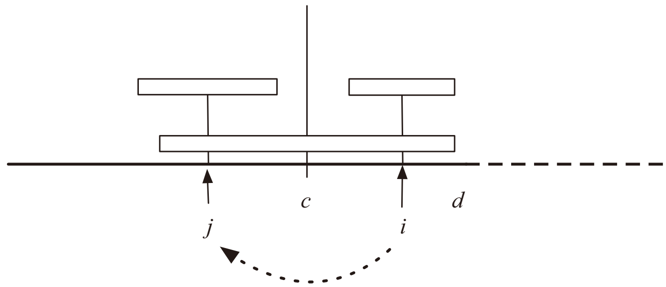


图 2.8　Manacher 算法。对于下标 <i，已计算出数组 p，现在要计算 $p[i]$。这是一个以 c 为中心的回文子串，其半径为 d-c，最大值为 d，而且 j 是 i 以 c 为对称的镜像。对称来看，以 j 为中心、$p[j]$ 为半径的回文子串应当与以 i 为中心的字符等同，至少在半径 d-i 内是如此。于是，$p[j]$ 对于 $p[i]$ 的值来说就是一个下界

```
def manacher(s):
    assert '$' not in s and '^' not in s and '#' not in s
    if s == "":
        return(0, 1)
    t = "^#" + "#".join(s) + "#$"
    c = 0
    d = 0
    p = [0] * len(t)
    for i in range(1, len(t) - 1):
        #                       -- 相对于中心 c 翻转下标 i
        mirror = 2 * c - i        # = c - (i-c)
        p[i] = max(0, min(d - i, p[mirror]))
        #                       -- 增加以 i 为中心的回文子串的长度
        while t[i + 1 + p[i]] == t[i - 1 - p[i]]:
            p[i] += 1
        #                       -- 必要时调整中心点
        if i + p[i] > d:
            c = i
            d = i + p[i]
    (k, i) = max((p[i], i) for i in range(1, len(t) - 1))
    return((i - k) // 2, (i + k) // 2)  # 输出结果
```

- **应用**

一个人在城里漫步，他的智能手机记录下了所有移动路线。我们获取这些路线记录，并尝试在其中找到某段特定路程，即在两点间往返的相同路程。为解决这个问题，可以提取一个所有路口的列表，并在其中寻找回文子串。

第3章　序列

什么是动态规划？把问题解答方案的所有子方案保存下来并将之合并，以此表示完整的解决方案，这种方法就是动态规划。我们用扫描方式计算子方案，将之保存起来以便后续使用（即“记忆化”原理）。这种技术在序列的相关问题中尤其奏效，因为序列问题的子问题有时可以用序列本身的前缀来定义。

3.1　网格中的最短路径

- **定义**

在一个 $(n+1)\times(m+1)$ 的网格中，每个格子的编号都是 (i, j)，其中 $0 \leqslant i \leqslant n$ 且 $0 \leqslant j \leqslant m$。格子之间用有权重的路径连接：每个格子 (i, j) 的来路格子都是 $(i-1, j)$、$(i-1, j-1)$ 和 $(i, j-1)$，只有第一行第一列的格子没有来路节点（图 3.1）[1]。问题旨在找到从 $(0, 0)$ 到 (n, m) 的最短路径。

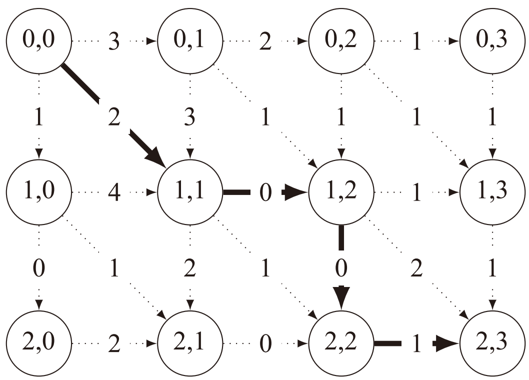


图 3.1　在一个有向网格中的最短路径，用加粗实线条表示

[1] 来路节点的定义为：要抵达一个格子 (i, j)，只能从其左方、上方和左上方进入；或者说，在网格中移动的方向只能是向右、向下或是向右下方。——译者注

- **时间复杂度为 $O(nm)$ 的算法**

图 3.1 有方向却没有环，即有向无环图（Directed Acyclic Graph，DAG），我们可以采用动态规划的思路，通过特定顺序计算从 (0, 0) 到每一个格子 (i, j) 的距离，来解决这个问题。首先，通向第一行和第一列的每个节点的距离是最容易计算的，因为通向每个节点的路径是唯一的。然后，用词典序方式计算通向所有 (i, j)，且有 $1 \leqslant i \leqslant n$ 且 $1 \leqslant j \leqslant m$ 的格子的距离。因此，从 (0, 0) 到 (i, j) 的距离一定有三种可能性，而这三个数字都是确定的，因为通向来路节点的最短距离都已经计算好了[①]。

- **变种**

很多经典问题都可以简化为这个简单问题，比如下节要介绍的各种问题。

3.2　编辑距离（列文斯登距离）

输入：AUDI, LADA

输出：LA–DA

–AUDI

3 个操作：删除 L，插入 U，把 A 替换成 I[②]

- **定义**

给定两个序列 x 和 y，需要多少次增、删、改的操作，才能把 x 变成 y？在 `unix` 命令 `diff` 中，这段距离显示为两个给定文件中的对应行之间相互变换所需的最少操作次数。

- **时间复杂度为 $O(nm)$ 的算法**

对 $n = |x|$，$m = |y|$ 使用动态规划法（图 3.2），算法时间复杂度为 $O(nm)$。我们要计算一个数组 $A[i, j]$，它是长度为 i 的前缀 x 和长度为 j 的前缀 y 之间的距离。我们从初始化开始，定义 $A[0, j] = j$ 和 $A[i, 0] = i$[③]。一般情况下，当 i 和 j 都 $\geqslant 1$，前缀的最后几个字母有三种可能情况：x_i 被删除；y_i 被插入到尾部；x_i 被 y_i 替换（如果它们不相同）。这三种情况让我们可以用递归方式定义以下三个公式

$$A[i,j] = \min\begin{cases} A[i-1, j-1] + \text{match}(x_i, y_j) \\ A[i, j-1] + 1 \\ A[i-1, j] + 1 \end{cases}$$

① 由于从左上角到右下角的前进方向只有三种可能性，即到达一个节点的来路只有三种可能，因而到达某个特定节点的距离仅由其三个可能的来路节点加上这三个来路节点到这个节点的距离决定，其中最短距离就是到达这个节点的最短路径。——译者注

② AUDI 和 LADA 是两个汽车品牌奥迪和拉达。——译者注

③ 长度为 0 的前缀变成长度为 n 的前缀需要的操作一定是 n 次。——译者注

其中 match 是一个返回布尔值的函数，当两个参数**不**相等时，函数会返回 1。这个方法定义了替换一个字母的成本。成本是可以调整的，比如说，成本可能取决于字母在键盘上的距离。

- **操作序列**

除了计算编辑距离，还要计算把 x 变换成 y 所需的操作次数。我们可以使用在图中查找最短路径的方式。通过遍历所有来路节点的距离，我们就能找到通向顶端的一条最短路径。这样一来，我们就能从节点 (n, m) 一直上升到 $(0, 0)$，并在上升的沿途路径中，计算出最优方案的所有操作步骤。最后，只要把这个序列逆序排列即可得到答案。

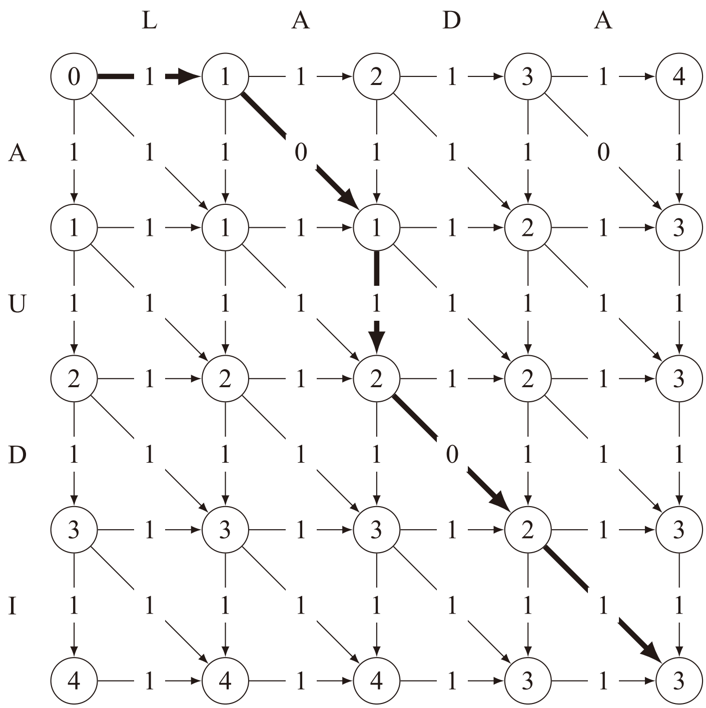


图 3.2　在一个有向图中的最短路径，它定义了两个单词之间的编辑距离[①]

- **实现细节**

在动态规划中，图的下标从 0 开始，而两个待变换字符串的下标从 1 开始。在实现的时候要注意这一点。

① 注意，竖排字符串是 AUDI，横排字符串是 LADA。从左上到右下的操作是：删除 L，距离 1；A 不变，距离 0；插入 U，距离 1；D 不变，距离 0；A 替换成 I，距离 1。最短路径的 5 个步骤是 1-0-1-0-1。——译者注

```
def levenshtein(x, y):
    n = len(x)
    m = len(y)
    #   初始化第 0 行和第 0 列
    A = [[i + j for j in range(m + 1)] for i in range(n + 1)]
    for i in range(n):
        for j in range(m):
            A[i + 1][j + 1] = min(A[i][j + 1] + 1,                 # 插入
                                  A[i + 1][j] + 1,                 # 删除
                                  A[i][j] + int(x[i] != y[j]))     # 替换
    return A[n][m]
```

- **深入思考**

在实际应用中，人们已经提出了性能更好的算法。比如，如果已知两个字符串编辑距离的长度上限 s，我们可以把上述动态规划，矩阵 A 的对角线长度限制为最大编辑距离 s，从而得到一个时间复杂度为 $O(s \min\{n, m\})$ 的算法（见参考文献 [28]）。

3.3　最长公共子序列

输入：GAC，AGCAT

```
输出：A  G  C  A  T
         |     |
         G     A  C
```

- **定义**

设一个符号集合∑。对于两个序列 $s, x \in \sum^\star$，如果存在下标 $i_1 < \cdots < i_{|s|}$ 使得对于所有 $x_{i_k} = s_k$，且其中 $k = 1, \cdots, |s|$，那么我们定义 s 是 x 的子序列。假设有两个序列 $x, y \in \sum^\star$，需要找到长度最大的子序列 $s \in \sum^\star$，而且它同时是 x 和 y 的子序列。

问题的另外一个表述方式为“配对”（见 9.1 节）。我们在两个序列 x 和 y 中寻找配对的最大可能性，使得这两个序列中的字母配对连线不交叉（图 3.3）。

- **应用**

在一个文件同步系统中，为了最小化网络传输流量，我们希望仅发送被修改过的部分，而不是把文件完整地发送到服务器。为了满足这个需求，必须找到旧文件和新文件的最大公共子序列。

这类问题同样出现在生物信息学中，用于对齐两条 DNA 序列。

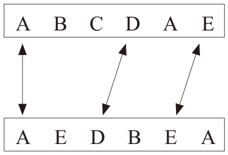


图 3.3 最长子序列问题可以被视为寻找两个给定序列的最大无交叉配对问题。比如，图中字母 B 的配对会与字母 D 的配对产生交叉

- **时间复杂度为 $O(nm)$ 的算法**

当 $n = |x|$ 和 $m = |y|$ 时，对于所有 $0 \leqslant i \leqslant n$ 和 $0 \leqslant j \leqslant m$，我们计算前缀 $x_1 \ldots x_i$ 和 $y_1 \ldots y_j$ 的最长公共子序列。这就得到一个复杂度为 $n \cdot m$ 的子问题。基于 $(i-1, j)$、$(i, j-1)$ 和 $(i-1, j-1)$ 在常数时间内得到的解，就能得到 (i, j) 的最优解。因此，我们可以在时间 $O(nm)$ 内解决问题 (n, m)。算法基于以下测试结果。

- **关键测试**

设序列 $x_1, \cdots, x_i$ 和 $y_1, \cdots, y_j$ 的最长公共子序列为 $A[i, j]$。在 $i = 0$ 或 $j = 0$ 时，$A[i, j]$ 为空。当 $x_i \neq y_j$ 时，x_i 和 x_j 中的一个肯定不在最优解中，而且 $A[i, j]$ 是 $A[i-1, j]$ 和 $A[i, j-1]$ 中最长的序列[①]。当 $x_i = y_j$ 时，存在一个最优解使得字符相关，而且 $A[i, j]$ 等于 $A[i-1, j-1] \cdot x_i$。这里的符号“·”表示字符串拼接。使用 maximum 函数可以让最长序列延伸[②]。

```
def longest_common_subsequence(x, y):
    n = len(x)
    m = len(y)
    #                              -- 计算最优长度
    A = [[0 for j in range(m + 1)] for i in range(n + 1)]
    for i in range(n):
        for j in range(m):
            if x[i] == y[j]:
                A[i + 1][j + 1] = A[i][j] + 1
            else:
                A[i + 1][j + 1] = max(A[i][j + 1],  A[i + 1][j])
    #                         -- 输出结果
    sol = []
    i, j = n, m
    while A[i][j] > 0:
        if A[i][j] == A[i - 1][j]:
            i -= 1
```

① 因为是两个序列比较，所以是比长短而不是大小。——译者注

② 当 $x_i = y_j$ 时，如果两个序列最后一个元素不一样，那么这两个元素中肯定有一个不在最终结果中，而且最优解一定等于各少一个元素的子序列中较长的那一个：$A[i\text{-}1, j]$ 是从 $A[i, j]$ 去掉 x_i，$A[i, j\text{-}1]$ 是从 $A[i, j]$ 去掉 y_j。这里的思路也是把大问题拆分成小问题：想找到 $A[i, j]$，可以先尝试找 $A[i\text{-}1, j]$ 和 $A[i, j\text{-}1]$，一步步缩短目标序列，从而简化问题。——译者注

```
        elif A[i][j] == A[i][j - 1]:
            j -= 1
        else:
            i -= 1
            j -= 1
            sol.append(x[i])
    return ''.join(sol[::-1])          # 列表反转
```

- **变种：给定多个序列**

假设我们不是从两个序列而是从 k 个序列中寻找最长公共子序列，序列长度分别为 $n_1, \ldots, n_k$，那么可以使用以下方法。我们需要计算一个 k 维矩阵 A，计算所有给定序列的前缀组合产生的最长公共子序列。这个算法的时间复杂度是 $O(2^k \prod_{i=1}^{k} n_i)$ 。

- **变种：给定两个排好序的序列**

当两个序列都已经排好序时，问题可以在时间 $O(n+m)$ 内解决。因为在这种情况下，我们可以使用合并两个已排序队列的方法（见 4.1 节）。

- **实践**

使用 BLAST 算法（Basic Local Alignment Search Tool），但它不能保证总是得出最优解。

3.4　升序最长子序列

- **定义**

给定一个包含 n 个整数的序列 x，需要找到它的一个子序列 s，使得 s 长度最长且是严格的升序。

- **应用**

想象有一条通向大海的直路，路边有很多房子，每一栋房子都有多层。当任何一栋房子和大海之间的所有房子的层数都能少一点的时候，从这栋房子就可以看到大海。我们希望所有房子都能够看到大海，同时只拆除尽可能少的房子来达到这个目标（图 3.4）



图 3.4　拆除尽可能少的房子，使得被保留下来的所有房子都能看到大海

● **复杂度为 $O(n\log n)$ 的算法**

准确地说，算法的复杂度是 $O(n\log m)$，其中 m 是最终计算得到的长度。当使用穷举算法时，对于每一个 i，我们希望为 x_i 元素拼接一个前缀为 $x_i,\cdots,x_{i-1}$ 的升序最长子序列。但是，这些子序列中哪个能得到最优解呢？让我们先考虑一下前缀的所有升序子序列。在一个升序子序列 y 中，两个属性是最重要的：长度，及其最后一个元素。从直觉上判断，在这些升序子序列中，我们更喜欢长度较长的，因为长度才是需要优化的属性；同时，用一个小元素结尾更容易达成目标。

为了证实这个直觉判断，我们把子序列 y 的长度记为 $|y|$，把 y 的最后一个元素记为 y_{-1}。当 $|y| \geqslant |z|$ 且 $y_{-1} \leqslant z_{-1}$，且两个不等式中有一个是严格不等时，我们称 y **支配** z。这时，只需要关注非支配子序列，将它补全为一个最优子序列（图 3.5）。

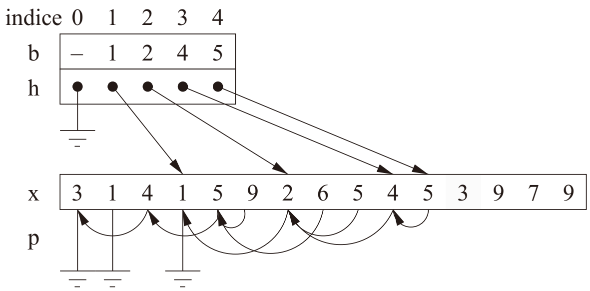


图 3.5 计算升序最长子序列。图中灰色部分是序列 x 正在被处理的部分。对于被考虑的前缀，被维护的的序列是 (),(1),(1, 2),(1, 2, 4),(1, 2, 4, 5)。处理输入字符 3 之后，序列 (1, 2, 4) 会被序列 (1, 2, 3) 取代

前缀 $x_i,\cdots,x_{i-1}$ 的各个非支配子序列的长度不同。对于长度 k，我们保存一个长度为 k 且以一个最小整数结尾的子序列。具体来说，我们维护一个数组 b，$b[k]$ 是长度为 k 的最长子序列的最后一个元素，且设定 $b[0]=-\infty$。

数组 b 是严格递增的。这样一来，在处理 x_i 元素时，最多只有一个子序列需要更新。尤其，当 k 满足 $b[k-1] < x_i < b[k]$ 时，我们可以用 x_i 补全长度为 $k-1$ 的序列尾部，以此得到一个更好的、长度为 k 的子序列，也就是说，该子序列的结尾是一个最小元素。当 x_i 比 b 中的所有元素都大时，我们使用 x_i 元素来增加 b。这是唯一可能的优化手段，并且使用二分法可以在在时间 $O(\log|b|)$ 内实现查找下标 k，其中 $|b|$ 是 b 的长度[①]。

① 替换和补全方式就是前面所说的“关注非支配子序列”，并将非支配子序列优化为最优子序列的具体过程。当 k 满足 $b[k-1]<x_i<b[k]$ 时，$|y|$-$|x|$ 且 $y_{-1}\leqslant z_{-1}$，因为 $b[k]$ 一定大于 $b[k-1]$。那么，当发现 x_i 时，我们只需更新 $b[k-1]$ 或 $b[k]$ 中的一个就有机会得到一个更好的答案，即长度更长的 $b[k-1]$ 或末尾元素更小的 $b[k]$。——译者注

● **实现细节**

数组 h 和 p 编成的列表将子序列编码 [①]。列表头使用 h 来表示，有 $b[k] = x[h[k]]$。元素 j 之前的元素用 $p[j]$ 来表示。列表使用常数 `None` 来结尾（图 3.5）。

```
from bisect import bisect_left

def longest_increasing_subsequence(x):
    n = len(x)
    p = [None] * n
    h = [None]
    b = [float('-inf')]                    #  负无穷
    for i in range(n):
        if x[i] > b[-1]:
            p[i] = h[-1]
            h.append(i)
            b.append(x[i])
        else:
            #   -- 二分查找：b[k - 1] < x[i] <= b[k]
            k = bisect_left(b, x[i])
            h[k] = i
            b[k] = x[i]
            p[i] = h[k - 1]
    # 显示结果
    q = h[-1]
    s = []
    while q is not None:
        s.append(x[q])
        q = p[q]
    return s[::-1]
```

● **变种：非降序子序列**

如果子序列不一定要严格升序，而只需不是降序即可，那么我们不再查找使得 $b[k-1] < x[i] \leqslant b[k]$ 成立的 k，而是查找使得 $b[k-1] \leqslant x_i < b[k]$ 成立的 k。这种算法可以用 Python 语言的 `bisect_right` 方法来实现。

● **变种：公共最长升序子序列**

给定两个序列 x 和 y，我们希望找到它们的公共最长升序子序列。这个问题可以在立方时间内解决：首先从 y 的排序开始，借此得到一个序列 z，然后寻找 x、y 和 z 的一个公共序列。2005 年，有人发表了一个更好的、时间复杂度为 $O(|x| \cdot |y|)$ 的算法（见参考文献 [29]），但这个算法复杂度早在 2003 年的 ACM/ICPC/NEERC 编程竞赛中就已经出现。

① 存储于 $b[\ ]$ 中长度不同的子序列。——译者注

3.5 两位玩家游戏中的必胜策略

- **定义**

假设两个玩家用一堆正整数进行游戏（图 3.6）。玩家 0 先开始。如果栈已经为空，那么他就输了游戏；否则，如果栈的顶端包含一个整数 x，他就可以选择去掉栈顶的一个元素或 x 个元素——后一个选项仅在栈中至少还有 x 个元素时被允许。然后，由玩家 1 来继续游戏。接下来，双方按照同样的规则继续游戏。现在有一个包含 n 个整数的栈 P，问题是：玩家 0 是否有一个必胜策略？也就是说，他是否能在玩家 1 做出任何选择的情况下都能确保获胜？

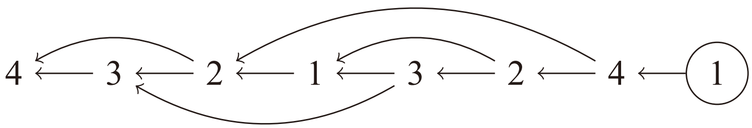


图 3.6 一局游戏

算法复杂度：线性。

- **动态规划的算法**

模拟游戏进程的成本非常高："二选一"会让方案组合成爆炸式增长。这里正是动态规划的用武之地。设一个大小为 n 的布尔型数组 G，$G[i]$ 表示在数字栈缩小为前 i 个元素的情况下，一个玩家能否在先手时获胜。最终目的是：玩家 0 找到一个必胜策略，使得玩家 1 处于无法获胜的劣势。

有一个基础情况是 $G[0]$ = `False`, 而对于 $i > 0$ 的情况，则有：

$$G[i]=\begin{cases}\overline{G[i-1]}\vee\overline{G[i-P[i]]} & \text{如果}P[i]\geqslant 0\\ \overline{G[i-1]} & \text{否则}\end{cases}$$

使用一个简单线性数字步数的遍历，就能填充数组 G，并通过寻找 $G[n-1]$ 的答案来解决问题。

第 4 章　数组

数组是最重要的基础数据类型之一。在很多简单问题中，除了数组之外不需要其他任何数据结构。我们可以在常数时间内修改一个指定下标的元素。相反，列表不能插入或删除元素，除非在线性时间内新建一个数组。数组中的元素从 0 开始索引。一定要注意，在读和写的时候，万一元素编号从 1 开始，在读取和显示其中数据时，仍然要从 0 开始[①]。

数组也能被简单用于编码一棵二叉树。一个下标为 i 的节点的父节点下标是 $[i/2]$，其左子节点下标是 $2i$，右子节点下标是 $2i+1$，根节点下标是 1。下标是 0 的元素被忽略掉了。

本章将探讨数组的经典问题，以及用来处理被称为“区间”的下标区间问题的数据结构，比如计算一个区间中最小值。最后两小节描述了动态的数据结构，提出了修改和查询数组的方法。

注意：**在 Python 语言中，可以用下标 –1 访问数组 t 的最后一个元素。数组的逆序副本可以通过 t[::–1] 来获得。在末尾新增一个元素的方法是 `append`。因此，Python 语言的数组类似 Java 语言的 `ArrayList` 类或者 C++ 语言的 `vector` 类。**

4.1　合并已排序列表

- **定义**

给定两个已排序的列表 x 和 y，我们希望生成一个有序的列表 z，包含 x 和 y 的所有元素。

- **应用**

这个操作在归并排序算法中很有用。对一个数组进行排序时，我们把它拆分为两个长度一样的部分（当数组有奇数个元素时，两部分相差一个元素）；同时，用递归方式对两部分进行排序；然后，再用下述过程把两部分合并起来。算法的时间复杂度是 $O(n\log n)$ 次比较。

① 如果问题描述的时候元素编号从 1 开始，程序编写的时候仍要从 0 开始为元素编号。——译者注

- **时间复杂度为线性的算法**

用连比法遍历两个列表，并逐步建立起 z。关键是每次都要从元素最小的列表中取值，这样就能保证结果一定是有序的。

```
def merge(x, y):
    z = []
    i = 0
    j = 0
    while i < len(x) or j < len(y):
        if j == len(y) or i < len(x) and x[i] < y[j]:
            z.append(x[i])
            i += 1
        else:
            z.append(y[j])
            j += 1
    return z
```

- **变种：*k* 个列表的合并**

为了快速找到需要取值的列表，我们可以把当前每个列表的 k 个元素存储在一个优先级队列中。算法的时间复杂度是 $O(n\log k)$，其中 n 是生成列表的长度。

4.2 区间的总和

- **定义**

每次查询都用下标区间 $[i, j)$ 来表示，而且需要返回在下标 i（包括）和下标 j（不包括）之间的区间中所有元素值 t 的总和。

- **每次查询时间复杂度为 $O(1)$ 的数据结构和时间复杂度为 $O(n)$ 的初始化方法**

只需计算一个包含了所有 t 的前缀且大小为 n+1 的数组。实际上，$s[j]=\sum_{i<j} t[i]$。特别是当 $s[0]=0$，$s[1]=t[0]$ 时，$s[n]=\sum_{i=0}^{n-1} t[i]$。那么查询 $[i, j)$ 的结果是 s[j]-s[i]。

4.3 区间内的重复内容

- **定义**

每个查询都用下标区间来表示，如 $[i, j)$，我们需要找到一个在区间内数组 t 中至少出现两次的元素 x，或声明所有元素各不相同。

- **每次查询时间复杂度为 $O(1)$ 的数据结构和时间复杂度为 $O(n)$ 的初始化方法**

通过一次从左到右的遍历，我们建立起一个数组 p，其中对于每个 j，当 $t[i]=t[j]$ 时，满足区间最大下标 $i < j$。当 $t[j]$ 第一次出现时，设 $p[j] = -1$。

为了计算 p，在遍历中，最后一次在 t 中出现的 x 的下标都要储存。如果能够保证 t 中元素的区间，我们可以用一个数组来存储；否则，需要用一个基于散列表的字典来存储。

同时，对于所有 $i \leqslant j$ 的情况，我们需要在一个数组 q 中保存每个 j 的最大值 $p[i]$。因此，为了响应查询 $[i, j)$，一旦有 $q[j-1] < i$，那么 $[i, j)$ 中所有元素都会各不相同；否则，$t[q[j-1]]$ 是一个重复值。

为了确定出现两次的下标，只需同时使用 $q[j]$ 计算下标 i，得到下标 i 的最大值。

4.4　区间的最大总和

- **定义**

这一静态问题基于一个值为 t 的数组，对于所有满足 $i \leqslant j$ 的下标对 $[i, j]$，需要计算 $t[i]+t[i+1]+, \cdots, +t[j]$ 的最大值。

- **复杂度为 $O(n)$ 的算法**

这一动态规划的算法是 Jay Kadane 在 1984 年发现的。对于每个下标 j，我们在所有满足 $0 \leqslant i \leqslant j$ 的下标中查找 $t[i]+, \cdots, +t[j]$ 的最大值。用 $A[j]$ 来记录这个值。这个值要么是 $t[j]$，要么由 $t[j]$ 和记为 $t[i]+, \cdots, +t[j-1]$ 的总和组成，该总和应该最大。于是有 $A[0] = t[0]$，因为对于 $j \geqslant 1$，这是唯一的可能性。综上所述，我们得到一个循环算式：$A[j] = t[j] + \max\{A[j-1], 0\}$。

- **变种**

问题可以推广到矩阵。给定一个维度为 $n \times m$ 的矩阵 M、一个行下标的区间 $[a, b]$ 和一个列下标的区间 $[i, j]$，就此定义一个矩阵中的矩形 $[a, b] \otimes [j, i]$。我们要找的是总和值最大的矩形。为此，只需在所有行下标的区间 $[a, b]$ 中循环，得到一个长度为 m 的数组 t，如 $t[i] = M[a, i]+,\cdots,+M[b, i]$。利用与行 $[a, b-1]$ 相关的数组，只需增加矩阵 M 的第 b 行，即可在时间 $O(m)$ 内得到数组 t。在上述算法的帮助下，我们可以在时间 $O(m)$ 内找到一个组成矩形总和值最大的列区间 $[i, j]$，矩形是由 $[a, b]$ 和 $[j, i]$ 来描述的。这样就能得到一个时间复杂度为 $O(n^2m)$ 的解决方案。

4.5　查询区间中的最小值：线段树

- **定义**

我们希望维护一个数据结构，该数据结构中存储了一个包含 n 个元素的数组 t，并能执行以下操作：

— 变更给定 i 值的 $t[i]$ 元素的值；

— 对于给定的集合下标 i 和 k，计算 $_{i\leqslant j<k}t[j]$ 最小值。

- **变种**

不需要太大变化，我们可以确定最小元素的下标，而非获取它的值。

- **每次查询时间复杂度为 $O(\log n)$ 的数据结构**

思路是用一棵二叉树（又称线段树）把数组 t 补全。每个节点代表数组 t 中的一个下标区间（图 4.1）。区间的大小是 2 的幂，一个节点的两个子节点代表区间左右两个半区。树的最底层保存了数组 t 中的元素值。在每个节点中，仅保存与节点相关区间内的数组最小值。

更新一次数组需要对树结构进行指数次更新，并作用于通向相关节点路径上的每个节点。在一个给定区间 $[i,k)$ 中查找最小值，是通过递归遍历来实现的。函数 `_range_min(j,start,span,i,k)` 返回数组 t 在区间 [start, start+span] ∩ $[i, k]$ 中的最小值，其中 j 是与该区间相关的节点的下标。终止查找有两种可能的条件：要么是当前节点的区间包含在 $[i,k)$ 范围内，这时需要返回节点的值；要么是当前节点的区间与 $[i,k)$ 不相交，这时需要返回 $+\infty$。

<table>
<tr><td colspan="15">1</td></tr>
<tr><td colspan="8">1</td><td colspan="7">5</td></tr>
<tr><td colspan="4">1</td><td colspan="4">2</td><td colspan="4">5</td><td colspan="3">∞</td></tr>
<tr><td colspan="2">1</td><td colspan="2">8</td><td colspan="2">3</td><td colspan="2">2</td><td colspan="2">5</td><td colspan="2">∞</td><td colspan="2">∞</td><td>∞</td></tr>
<tr><td>3</td><td>1</td><td>9</td><td>8</td><td>3</td><td>4</td><td>2</td><td>7</td><td>5</td><td>∞</td><td>∞</td><td>∞</td><td>∞</td><td>∞</td><td>∞</td></tr>
</table>

图 4.1 用于在一个区间中确定最小值的树形数据结构

- **对复杂度的分析**

为了证明查询时间为 $O(\log n)$，我们要区分遍历过程中经过的 4 种节点。假设 $[s,t)$ 是节点相关的区间：

— 当 $[s,t)$ 和 $[i,k)$ 不相交时，节点被称为**空节点**；

— 当 $[s,t)\subseteq[i,k)$ 时，节点被称为**满节点**；

— 当 $[s,t)\supset[i,k)$ 时，节点被称为**严格节点**；

— 否则，节点被称**重叠节点**。

注意，对于一个重叠节点来说，$[s,t)$ 和 $[i,k)$ 的区间互相重叠，但并不互相包含。分析方法就是确定每个类型的节点数量。

查找方法 `range_min` 遍历一个对数数量级的满节点，它们对应着 $[i,k)$ 区间内不相交的区间分量。对于空节点也一样，空节点对应着 $[i,k)$ 区间的补集分量。当检测到一个重叠节点的子节点总是满节点或空节点，并且只有根节点是严格节点的时候，分析结束。

```
class RangeMinQuery :
    def __init__(self, t, INF=float('inf')):
        self.INF = INF
        self.N = 1
        while self.N < len(t):                      # 找到数组大小 N
            self.N *= 2
        self.s = [self.INF] * (2 * self.N)
        for i in range(len(t)):                     # 把 t 存入叶子节点
            self.s[self.N + i] = t[i]
        for p in range(self.N - 1, 0, -1):          # 填充所有节点
            self.s[p] = min(self.s[2 * p], self.s[2 * p + 1])

    def __getitem__(self, i):
        return self.s[self.N + i]

    def __setitem__(self, i, v):
        p = self.N + i
        self.s[p] = v
        p //= 2                                     # 在树上上移一层
        while p > 0:                                # 更新节点
            self.s[p] = min(self.s[2 * p], self.s[2 * p + 1])
            p //= 2

    def range_min(self, i, k):
        return self._range_min(1, 0, self.N, i, k)

    def _range_min(self, p, start, span, i, k):
        if start + span <= i or k <= start:         # 不相交区间
            return self.INF
        if i <= start and start + span <= k:        # 包含区间
            return self.s[p]
        left = self._range_min(2*p,     start,            span//2, i, k)
        right = self._range_min(2*p + 1, start + span // 2, span//2, i, k)
        return min(left, right)
```

4.6　计算区间的总和：树状数组（Fenwick 树）

- **定义**

我们希望维护一个数据结构，它保存着包含 n 个值的数组 t，并可以进行下列操作：

— 对于给定下标 i，更新 $t[i]$ 的值；

— 对于给定下标 i，计算 $t[1]+\cdots+t[i]$。

出于技术原因，t 的下标范围是从 1 到 n–1，并不包含 0。

- **变种**

仅需一个小改动，这个数据结构同样可以被用于在时间 $O(\log n)$ 内执行下列运算：

— 对于给定的下标 a 和 b，为集合中每个区间 $t[a], t[a+1],\cdots, t[b]$ 增加一个值；

— 对于给定下标 i，获取 $t[i]$ 的值。

如前一节介绍过的方法，此问题可以使用一个线段树来解决。只需把其中的∞替换为 0，并把 `min` 操作替换成加法操作，就可以得到一个每次查询时间复杂度为 $O(\log n)$ 的数据结构来解决问题。本节介绍的数据结构性能与之前的类似，但可以更快实现。

- **每次查询时间复杂度为 $O(\log n)$ 的数据结构（见参考文献 [6]）**

这个数据结构不再像前一节所述的那样原封不动地存储数组 t，而是存储 t 的区间总和。因此，我们需要新建一个数组 s，比如，使得满足 $j \in I(i)$ 的时候，有 $s[i]$ 是 $t[j]$ 的总和，其中 $\mathrm{I}(i)$ 是按下述方式定义的一个区间。各区间通过以下方式组织成一个树形结构，它们之间有两种关系：父节点和左相邻节点（图 4.2）。

— 当 $a \in \{0, 1\}^*$ 且 i 是 $a10^k$ 的二进制形式时，输入值 $s[i]$ 包含数组 t 在区间 $I(i) = \{a0^k1, \cdots, i\}$ 内的总和。

— 下标 $i = a10^k$ 的父节点是 $i+10^k$。

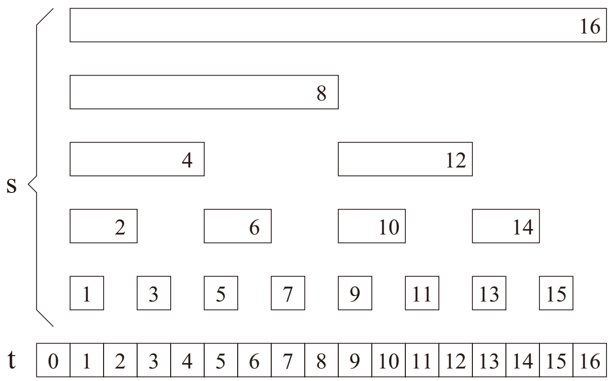
— 下标 $i = a10^k$ 的左相邻节点是 $j = a00^k$。区间 $\mathrm{I}(j)$ 是 $\mathrm{I}(i)$ 的左侧区间。

因此，当 $i = a10^k$ 时，前缀 $t[1]+\cdots+t[i]$ 的总和等于 $s[i]$ 加上前缀 $t[1]+\cdots+t[a00^k]$。这里需要使用递归计算。

在图 4.2 的例子里，对 $t[11]$ 更新就必须改变 $s[11=01011_2]$、$s[11=01011_2]$、$s[11=01011_2]$、$s[12=01100_2]$、$s[16=10000_2]$，而前缀 $t[1],\cdots, t[11]$ 的总和是 $s[11=01011_2]$、$s[10=01010_2]$、$s[8=01000_2]$。

实现上述结构的一个重要步骤是读取 i 的最低有效位，也就是说，把二进制格式的数字 $a10^k$ 转换成 10^k。这一操作可以使用 $i\&-i$ 实现。以下为解释：

$$
\begin{aligned}
i &= a10^k \\
\bar{i} &= \bar{a}01^k \\
-i = \bar{i} + 1 &= \bar{a}10^k \\
i \& -i &= 10^k
\end{aligned}
$$



i	二进制	父节点	左相邻节点
1	001	2	∅
2	010	4	∅
3	011	4	2
4	100	8	∅
5	101	6	4
6	110	8	4
7	111	8	6

图 4.2　一个树状数组的例子。“XX 是○○的父节点”，这一关系以从上垂直向下投影的方式体现，比如 8 是 4、6 和 7 的父节点。如果区间 $\mathrm{I}(i)$ 在 $\mathrm{I}(j)$ 的左侧邻接，如 4 是 5 和 6 的左相邻，那么下标 i 是 j 的左相邻节点。

```
class Fenwick :
    def __init__(self, t):
        self.s = [0] * len(t)
        for i in range(1, len(t)):
            self.add(i, t[i])

    def prefixSum(self, i):
        sum = 0
        while i > 0:
            sum += self.s[i]
            i -= (i & -i)
        return sum

    def intervalSum(self, a, b):
        return self.prefixSum(b) - self.prefixSum(a-1)

    def add(self,  i, val):
        assert i > 0
        while i < len(self.s):
            self.s[i] += val
            i += (i & -i)

    # 变种：

    def intervalAdd(self, a, b, val):
        self.add(a,     +val)
        self.add(b + 1, -val)

    def get(self, i):
        return self.prefixSum(i)
```

4.7　有 k 个独立元素的窗口

- **定义**

给定一个包含 n 个元素的序列 x 和一个整数 k，我们希望确定 $[i, j)$ 的所有最大区间，使得 $x_i,\cdots, x_{j-1}$ 都严格地由 k 个不同元素组成（图 4.3）。

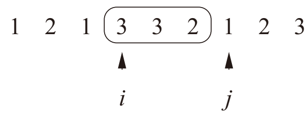


图 4.3　一个严格包含了两个不同元素的窗口

- **应用**

缓存就是被放在慢速读写内存之前的快速内存。慢速内存的寻址空间被分割成相同的大小，称

作**分页**。缓存空间是有限的，仅能保存 k 个分页。计算机随时间推移访问内存，由此形成一个序列 x，其中每个元素是一个被请求访问的分页。当一个被请求访问的分页在缓存中时，读写速度加快，否则称为**缓存未命中**。然后，应当查找慢速内存的一个分页，令其替代缓存中的另一个分页，并将后者移出缓存。在序列 x 中查找严格包含 k 个不同元素的区间，这一问题变为在假设缓存初始配置最优的情况下，找到那些不存在缓存未命中的时间区间。

- **时间复杂度为 $O(n)$ 的算法**

思路是使用两个游标 i 和 j 来遍历序列 x，其中 i 和 j 确定了窗口。我们借助出现频率计数器 occ，在一个变量 `dist` 中维护集合 $x_i,\cdots,x_{j-1}$ 中不同元素的数量。当 `dist` 超过了参数 k 值，我们让 i 前进，否则让 j 前进[①]。以下实现返回了一个迭代器。该实现也可以用于另一个函数，单独处理每个区间。由于两个游标 i 和 j 只能前进，因而最多只能执行 $2n$ 次操作，从而获得一个线性的复杂度。

```
def windows_k_distinct(x, k):
    dist, i, j = 0, 0, 0              # dist = |{x[i],..,x[j -1]}|
    occ = {xi: 0 for xi in x}         # 用 x[i:j] 表示的出现次数
    while j < len(x):
        while dist == k:              # 移动头部的区间
            occ[x[i]] -= 1            # 更新计数器
            if occ[x[i]] == 0:
                dist -= 1
            i += 1
    while j < len(x) and(dist < k or occ[x[j]]):
        if occ[x[j]] == 0:            # 更新计数器
            dist += 1
        occ[x[j]] += 1
        j += 1                # 移动末尾的区间
    if dist == k:
        yield(i, j)           # 发现一个区间
```

① 对于下标和遍历所使用的游标来说，前进就是下标 +1，或是迭代器获取下一个元素。——译者注

第5章　区间

与区间相关的很多问题都可以使用动态规划来解决。位于一个给定阈值之前和之后的区间可以形成两个独立的子实例。

如果问题允许，使用格式为 $[s, t)$ 的半开半闭区间会更简便，因为其中的元素数量更容易计算（仅需 $t-s$，其中 s 和 t 都是整数）。

5.1　区间树（线段树）

- **定义**

把 n 个给定区间存储在一个数据结构中，使得拥有下述格式的查询能快速返回结果：**对于一个给定值 p，哪个是所有包含 p 的区间列表？**我们先假设所有区间都是半开半闭的形式 $[l, h)$，但这个数据结构也适应其他区间形式。

- **每次查询时间复杂度为 $O(\log n+m)$ 的数据结构**

m 是返回的区间数量。这是一个二叉树结构，描述如下。设 S 是一个待存储区间的集合。我们选择一个满足下述条件的中间值 `center`。中间值 `center` 把所有区间分成三组：区间集合 L 在中间值 `center` 的左侧；区间集合 C 包含中间值 `center`；区间集合 R 在中间值 `center` 的右侧。那么，树形结构的根用递归方式保存着中间值 `center` 和 C，其左侧子树和右侧子树分别保存 L 和 R（图 5.1）。

为了能快速响应查询，集合 C 被以有序列表的形式存储。列表 `by_low` 保存了 C 中按开头排列顺序的区间；同时，列表 `by_high` 保存了 C 中按尾排列顺序的区间。

为了响应对 p 点的查询，只需要把 p 与中间值 `center` 比较：如果 $p >$ `center`，那么需要用递归方式在左侧子树中查找包含了 p 的区间，并在其中添加 C 中的区间 $[l, h)$，且满足 $l \leq p$。这是正确的作法，因为通过构建这些区间，满足 $h >$ `center`，也就满足了 $p \in [l, h)$。否则，如果

$p \geqslant$ center，则需要用递归方式在右侧子树中查找包含 p 的区间，并在其中添加 C 中的区间 $[l, h)$，且满足 $p < h$。

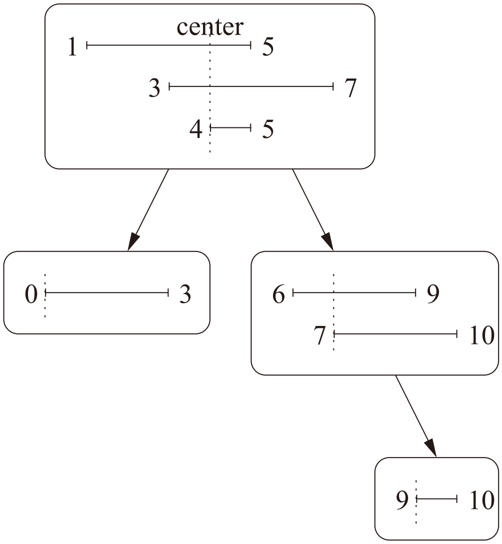


图 5.1　存储 7 个区间的树

- **选择中间值 `center`**

　　为了让二叉树能够平衡，我们可以选择中间值 `center` 作为要存储区间的正中元素。这样一来，一半区间就会被存进右侧子树，保证深度是指数级别的。与快速排序算法类似，如果中间值 `center` 是从区间中随机选择的，则期望性能也将是类似的[①]。

- **复杂度**

　　构建二叉树需要的时间复杂度为 $O(\log n)$，处理一个查询所需时间为 $O(\log n+m)$，其中指数部分源于对有序列表的二分查找。

- **实现细节**

　　在实现中，区间用 n 元组的方式呈现，其中排在最前面的两个元素保存着区间的首尾边界。其他元素可用于传输补充信息。

　　二分查找通过方法 `bisect_right(t,x)` 来实现，它返回了 i，使得当 $j > i$ 时有 $t[i] > x$。注意，不要使用子列表 `by_high[i:]` 在数组 `by_high` 中循环，因为创建长度为 `len(by_high)` 的子列表所用时间是线性的，这会把算法复杂度从 $O(\log n+m)$（m 是返回列表的大小）提升至 $O(\log n+n)$。

① 这和快速排序算法随机选择分隔点对性能的影响类似。——译者注

数组保存了数值对 (value, interval)，所以我们使用 bisect_high 方法来查找格式为 $(p, (\infty, \infty))$ 的一个元素 x 的插入点。

```
class _Node :
    def __init__(self, center, by_low, by_high, left, right):
        self.center = center
        self.by_low = by_low
        self.by_high = by_high
        self.left = left
        self.right = right

def interval_tree(intervals):
    # 下面的测试会降低性能
    # assert intervals == sorted(intervals)
    if intervals == []:
        return None
    center = intervals[len(intervals) // 2][0]
    L = []
    R = []
    C = []
    for I in intervals:
        if I[1] <= center:
            L.append(I)
        elif center < I[0]:
            R.append(I)
        else:
            C.append(I)
    by_low = sorted((I[0], I) for I in C)
    by_high = sorted((I[1], I) for I in C)
    IL = interval_tree(L)
    IR = interval_tree(R)
    return _Node(center, by_low, by_high, IL, IR)

def intervals_containing(t, p):
    INF = float('inf')
    if t is None:
        return[]
    if p < t.center:
        retval = intervals_containing(t.left, p)
        j = bisect_right(t.by_low, (p, (INF, INF)))
        for i in range(j):
            retval.append(t.by_low[i][1])
    else:
        retval = intervals_containing(t.right, p)
        i = bisect_right(t.by_high, (p, (INF, INF)))
        for j in range(i, len(t.by_high)):
            retval.append(t.by_high[j][1])
    return retval
```

5.2 区间的并集

- **定义**

给定一个包含 n 个区间的集合 S，我们希望确定具有多个非连接区间有序列表形式的并集 L，且 $\bigcup_{I\in S} = \bigcup_{I\in L}$。

- **使用扫描方式复杂度为 $O(n\log n)$ 的算法**

从左向右扫描区间临界值。在每个给定时刻，在 open 中维护开放区间（尚未看到结尾的区间）的数量。当该数量变为零后，需要在解中假设一个新区间 [open, x]，其中 x 是扫子的通常位置，open 是 open 变为正值的最后一个位置。

- **实现细节**

记录下被处理区间的临界值顺序。当区间是封闭或半开时，这个顺序是正确的。对于开放区间而言，需要在处理区间的起始值 (y, z) 之前处理区间的结束值 (x, y)。

```
def intervals_union(S):
    E = [(low, -1) for(low, high) in S]
    E += [(high, +1) for(low, high) in S]
    nb_open = 0
    last = None
    retval = []
    for x, _dir in sorted(E):
        if _dir == -1:
            if nb_open == 0:
                last = x
            nb_open += 1
        else:
            nb_open -= 1
            if nb_open == 0:
                retval.append((last, x))
    return retval
```

5.3 区间的覆盖

- **应用**

假设有一片平直的海滩，周围是数座如点一样小的岛屿，我们希望沿着海滩放置最小数量的天线，让信号覆盖所有岛屿。所有天线的信号覆盖半径都是 r。

- **观察**

如果在每个岛屿周围画出半径 r 的范围，我们就能在海滩上找出必须安装一个天线的区间。问题就简化为以下问题（图 5.2）。

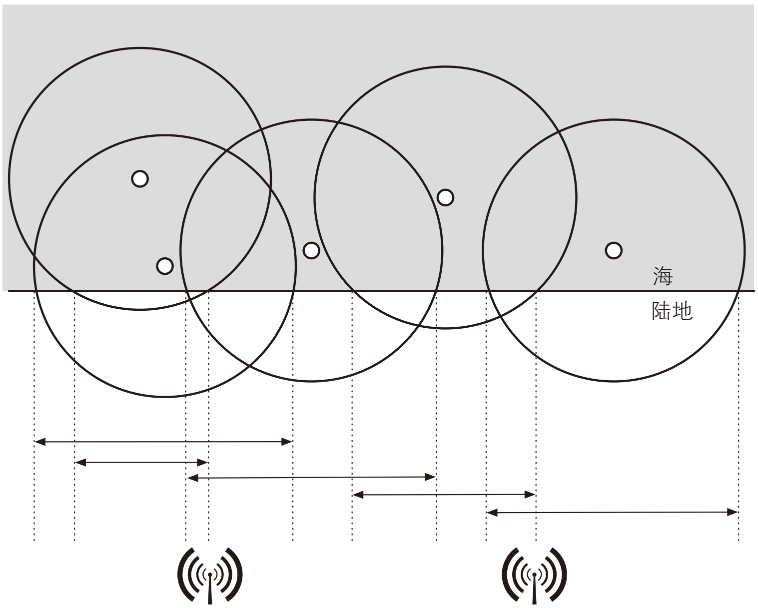


图 5.2 需要多少天线才能覆盖所有岛屿？问题简化为区间覆盖问题

- **时间复杂度为 $O(n\log n)$ 的算法**

使用扫描方式，我们从右侧开始按升序方向处理所有区间。我们维护一个解 S 来保存已经扫描过的区间，S 的最小值是 $|S|$，在相等极限情况下的值是 $\max S$。

算法很简单：如果对于一个区间 $[l, r]$ 有 $1 \leqslant \max S$，那就什么都不做；否则，就把 r 添加入 S。思路是需要想方设法覆盖 $[l, r]$，并且，通过选择可覆盖区间的最大值还能增加覆盖后续待处理区间的可能性。

```
def interval_cover(I):
    S = []
    for start, end in sorted(I, key=lambda v: (v[1], v[0])):
        if not S or S[-1] < start:
            S.append(end)
    return S
```

第6章　图

图是由顶点集合 V 和边集合 E 组成的对象。一般来说，边与两个不同的顶点对相关，并且边不区分方向，也就是说 (u, v) 和 (v, u) 表示的是同一条边。

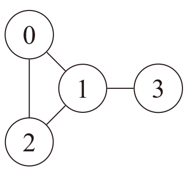
有时我们会考虑一个变种，即**有向图**，有向图的边是有方向的。在这种情况下，习惯上称边为**弧**，弧 (u, v) 的起点是 u，终点是 v。本章的大多数算法都基于有向图的操作，但也可能把边 (u, v) 替换成两条弧 (u, v) 和 (v, u)，借此应用于无向图。

图也可以包含额外的信息，比如在顶点或边上标注权重或字符。

6.1　使用 Python 对图编码

一个简单方法是使用从 0 到 n-1 的整数来区分 n 个顶点。但在输入文件里显示或编码的时候，通常从 1 开始计数。读者要注意，在读取和显示的时候，记得在下标中添加或删除 1 个下标。

图的边可以通过邻接数组或邻接矩阵这两种方式表示。邻接矩阵很容易实现，但更占空间。在此情况下，图将用一个二进制数的矩阵 E 来表示，其中 E[u, v] 代表存在弧 (u, v)（图 6.1）。




```
# 邻接数组
G = [[1,2],[0,2,3],[0,1],[1]]

# 邻接矩阵
G = [[0,1,1,0],
     [1,0,1,1],
     [1,1,0,0],
     [0,1,0,0]]
```

图 6.1　一个图及其可能的编码方式

邻接数组通过一个数组 G 来符号化一个图。对于顶点 u，$G[u]$ 是 u 的邻接顶点列表。我们同样

可以借助文本标识符来指示顶点。因此，G 可以是一个字典，其中的键是字符串，值是邻接顶点列表。例如，如果三角形由三个顶点 axel、bill 和 carl 组成，那就可以用以下字典来编码[①]。

```
{'axel':['bill','carl'], 'bill':['axel','carl'], 'carl':['axel','bill']}
```

本书中提到的算法通常以邻接数组为基础。

在有向图中，有时需要两个数据结构 G_out 和 G_in，包含每个节点离开的弧和进入的弧。但是，我们会储存弧的顶点而非弧本身。因此，对于任意顶点 u，$G_out[u]$ 包含每条离开的弧 (u, v) 的顶点数组 v，$G_in[u]$ 包含每条进入的弧 (v, u) 的顶点数组 v。

有一个简单方式用来保存顶点和边上的标签：使用按顶点索引的额外数组，或者按顶点对索引的矩阵。这样一来，图本身的编码结构 G 就不受影响，而 G 也可以在不被修改且不考虑标签的情况下用于代码之中。利用类的属性来表示顶点和边，这种方式对高效编程来讲就非常耗时了。

6.2 使用 C++ 或 Java 对图编码

由于 C++ 或 Java 标准库中的列表和字典的使用比较麻烦，而且速度也有点慢，因而我们提议使用一个高效的编码方式——链列。每个弧使用一个下标 e 来表示，那么 dest[e] 就是弧 e 的目标顶点。离开顶点 u 的弧将以下述方式组织成链列。列表中的第一条弧是 arc[u]，第二条弧是 succ[arc[u]]，第三条弧是 succ[succ[arc[u]]]，以此类推[②]，列表的结尾是特殊值 -1。对于没有离开弧的顶点 u，我们定义 arc[u]== -1。

```
const int MAX_NODES = 500;                          // 举例
const int MAX_ARCS  = 2*MAX_NODES*MAX_NODES;        // 举例

int nb_nodes = 0;
int nb_arcs  = 0;
int arc[MAX_NODES] = {0};
int succ[MAX_ARCS], dest[MAX_ARCS];

void clear_graph(int n) {
    nb_nodes = n;
    nb_arcs  = 0;
```

① 在邻接数组中，下标为 0 的元素值是集合 {1, 2}，说明元素 0 连接着元素 1 和 2，下标为 1 的元素值是集合 {0, 2, 3}，说明元素 1 连接着 0、2 和 3，其他元素以此类推。在邻接矩阵中，可以看到矩阵沿着从左上角到右下角的斜线呈对称，i 行 j 列等于 0，说明元素 i 和 j 不连通，等于 1 说明 i 和 j 连通。比如在第一行中，0 列 0 行的第 0 个元素 0 和自己不连通，1 列 0 行的第 0 个元素和第一个元素连通，0 列 2 行等于 1 说明 0 和 2 连通，0 列 3 行和 3 列 0 行都等于 0，说明 0 和 3 不连通，以此类推。——译者注

② succ 是单词 successor 的前缀，表示用一种递归方式定义了每个顶点及其一层层的后继顶点。——译者注

```
    for(int v=0; v<nb_nodes; v++)
        arc[v] = -1;
}

void add_arc(int u, int v) {
    succ[nb_arcs] = arc[u];
    dest[nb_arcs] = v;
    arc[u]  = nb_arcs++;
}

#define forall_neighbors(u, v) \
   for(int e=arc[u]; e!=-1 && (v=dest[e], 1); e=succ[e])
```

6.3 隐式图

有时，图以隐式方式给定，比如网格的格式，其中图的顶点是网格的一个个单元，而边由网格单元间的邻接关系来定义，这就像一个迷宫。相关对象是另一种隐式图，其中的弧对应着一个本地的改动（一个对象被改动后变成另外一个对象）。

- **例子：高峰期**

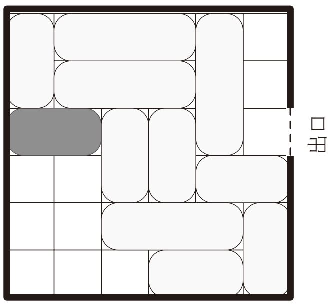


图 6.2 一局“高峰期”游戏

“高峰期”是一种能在商店里买到的智力游戏。棋盘是一个 6×6 的网格（图 6.2）。棋盘上的小轿车（长度为 2）和大卡车（长度为 3）停在网格中。车里没有司机，也不能离开网格的范围。其中一辆小轿车被特别标识为红色，玩家的目标是把这辆小轿车从网格侧面唯一一个出口挪出网格。为了实现目标，玩家可以向前或向后移动棋盘中的所有车辆[①]。

我们马上使用图来实现建模。在 k 辆车中，每一辆车都对应着一个固定模型和可变模型，固定模型由大小、方向和固定位置组成（比如一辆纵向行驶的车所在的列）；可变模型由自由坐标组成。所有自由坐标的向量完整地编码了网格的组态。遍历这张图的关键函数从一个向量开始，枚举了所

① 类似于中国的“华容道”游戏，通过移动迷宫中的长短块，把黑色块移出迷宫。——译者注

有通过一步移动能得到的向量组态。以下是图 6.2 中一局游戏的组态编码[①]。

```
orientat = [1,0,1,0,0,1,1,0,0,1,0]     # 0= 横向， 1= 纵向。      方向配置
longueur = [2,3,3,3,2,2,2,2,3,2,2]     # 2= 轿车， 3= 卡车         长度配置
coorfixe = [0,0,4,1,2,2,3,3,4,5,5]     # 固定坐标，即原始出发点
coorvari = [0,1,0,1,0,2,2,4,2,4,3]     # 可变坐标，移动后的坐标
rouge = 4                              # 红色小轿车的下标
```

比如，如果 orientat[i]=0，那么对于 coorvari[i] $\leqslant x <$ coorvari[i] + longueur[i] 以及 y = coorfixe[i]，小轿车 i 占用了所有 (x, y) 的格子。如果 orientat[i] = 1，那么对于 x = coorfixe[i] 以及 coorvari[i] $\leqslant y <$ coorvari[i] + longueur[i]，小轿车 i 占用了所有 (x, y) 的格子。

6.4　深度优先遍历：深度优先算法

- **定义**

深度优先遍历是一种对图的遍历方法，它从一个给定节点开始，以递归方式遍历该节点的相邻节点。深度优先算法的英文全称是 Depth-first search，简称 DFS 算法。

复杂度：算法的时间复杂度是 $O(|V|+|E|)$。

- **应用**

深度优先算法主要用于从图中找到一个给定节点能够到达的所有节点。这种遍历方式也是本书后续要介绍到的很多算法的基础，比如，找到图中的重连通分量或拓扑排序（见 6.7 节和 6.8 节）。

- **实现细节**

为了不重复遍历一个顶点的相邻节点，我们需要使用一个布尔型数组对已访问过的节点进行标注。

```
def dfs_recursive(graph, node, seen):
    seen[node] = True
    for neighbor in graph[node]:
        if not seen[neighbor]:
            dfs_recursive(graph, neighbor, seen)
```

- **更好的实现**

上述使用递归的实现方式不能处理较大的图，因为程序的调用栈是有限的。在 Python 语言中，`setrecursionlimit` 方法让我们能稍微突破一点限制，但总的来说，递归调用还是不能超过数千次。为了缓解这个问题、提高效率，可以采用迭代方式的实现。栈 `to_visit` 包含所有被发现但尚未处理的顶点。

---

① orientat 意为“方向”，longueur 意为“长度”，coorfixe 意为“固定坐标”，coorvari 意为“可变坐标”，rouge 意为“红色”。——译者注

```
def dfs_iterative(graph, start, seen):
    seen[start] = True
    to_visit = [start]
    while to_visit :
        node = to_visit.pop()
        for neighbor in graph[node]:
            if not seen[neighbor]:
                   seen[neighbor] = True
                   to_visit.append(neighbor)
```

- **网格的情况**

假设一个网格中某些格子是可以通过的（使用字符填充），而某些格子不能通过（使用 # 字符填充），就如同一个迷宫。从一个格子开始，我们可以抵达其周围相邻的 4 个格子；位于边缘的格子除外，因为其相邻格子较少。在下述实现中，我们借助这个网格，并用字符 *X* 标注已访问过的格子。为了优化可读性，我们使用了递归遍历方式。

```
def dfs_grid(grid, i, j, mark='X', free='.'):
    height = len(grid)
    width = len(grid[0])
    to_visit = [(i, j)]
    grid[i][j] = mark
    while to_visit:
         i1, j1 = to_visit.pop()
         for i2, j2 in[(i1 + 1, j1), (i1, j1 + 1),
                        (i1 - 1, j1), (i1, j1 - 1)]:
            if 0 <= i2 < height and 0 <= j2 < width and grid[i2][j2] == free:
               grid[i2][j2] = mark                        # 标注已访问状态
               to_visit.append((i2, j2))
```

6.5 广度优先遍历：广度优先算法

- **定义**

与其从当前节点开始遍历尽可能远的距离（**深度优先**），不如从一个起始节点开始，按距离升序顺序枚举一个图的所有节点（**广度优先**）。广度优先算法的英语全称为 Breadth-First Search，简称 BFS 算法。

- **关键测试**

我们从初始节点开始按照距离升序处理所有节点，因此，就需要一个能够维护这个顺序的数据结构。队列是一个不错的选项：如果对于每个被取出作为头部的顶点，我们都把其相邻节点添加到尾部，那么在任意情况下都能证明，该顶点在头部只包含距离为 d 的节点，在尾部仅包含距离为 d+1 的节点；只要距离为 d 的顶点在头部尚未被用完，那么在尾部就只有被添加的距

离为 $d+1$ 的顶点。

- **时间复杂度为线性的 $O(|V|+|E|)$ 的算法**

广度优先算法使用与深度优先算法相同的数据结构，但有两个差别：一是深度优先算法使用栈，而广度优先算法使用队列；二是在广度优先算法中，顶点只在被加入队列时被标注，在离开队列时不标注，否则，内存的使用将会达到一个二次方的复杂度。

- **实现细节**

广度优先算法的主要好处是，它能在一个给定的非加权图的数据源中确定距离。算法的实现计算了这些距离，以及在最短路径树形结构中的前驱顶点。距离数组同样也用于标注遍历过程中遇到的顶点。

```
from collections import deque

def bfs(graph, start=0):
    to_visit = deque()
    dist = [float('inf')] * len(graph)
    prec = [None] * len(graph)
    dist[start] = 0
    to_visit.appendleft(start)
    while to_visit:                                  # 一个空的队列值是“假”
        node = to_visit.pop()
        for neighbor in graph[node]:
            if dist[neighbor] == float('inf'):
                dist[neighbor] = dist[node] + 1
                prec[neighbor] = node
                to_visit.appendleft(neighbor)
    return dist, prec
```

6.6　连通分量

- **定义**

如果对于 A 中的顶点 u 和 v，存在一条从 u 到 v 的路径，那么图中满足 $A \subset V$ 的部分被称作**连通分量**。比如，我们可以计算一个图的连通分量。当图中只存在唯一一个连通分量的时候，就被称作**连通图**。

图 6.3 展示了用 ASCII Art 制作的 CleanBandit 乐队的标识 ①。它可以被视为一个用 # 字符表示顶点的图，而且，当且仅当两个顶点垂直或水平地互相接触时，这两个顶点才被一条边连接。这个图包含 4 个连通分量。

① ASCII Art 是一种使用 ASCII 字符（包含很多控制字符）拼接组合形成文字、图片和动画的艺术表现形式。——译者注

```
........................................    ........................................
...................#........#####.......    ...................1........22222.......
..................###......#######......    ..................111......2222222......
.................#####....#########.....    .................11111....222222222.....
................#######....#######......    ................1111111....2222222......
...............#########....#####.......    ...............111111111....22222.......
..............###########...............    ..............11111111111...............
.............#############..............    .............1111111111111..............
............###############.............    ............111111111111111.............
.............#############..............    .............1111111111111..............
..............###########...............    ..............11111111111...............
.........#.....#########................    .........3.....111111111................
........###.....#######...#########.....    ........333.....1111111...444444444.....
.......#####.....#####....#########.....    .......33333.....11111....444444444.....
......#######.....###.....#########.....    ......3333333.....111.....444444444.....
.....#########.....#......#########.....    .....333333333.....1......444444444.....
....###########...........#########.....    ....33333333333...........444444444.....
........................................    ........................................
```

图 6.3　对 Clean Bandit 乐队的标识图像执行算法前和执行算法后的网格状态[①]

- **应用**

假设桌子上放着一枚骰子，我们从垂直方向给骰子拍摄照片，并希望用简单方法确定反面的点数。为了达到目的，我们用灰度色阶将该图像色调分离（即减少颜色的数量），以此获取一个黑白图片，让骰子的每个点都对应图片中的一个连通分量（图 6.4）。

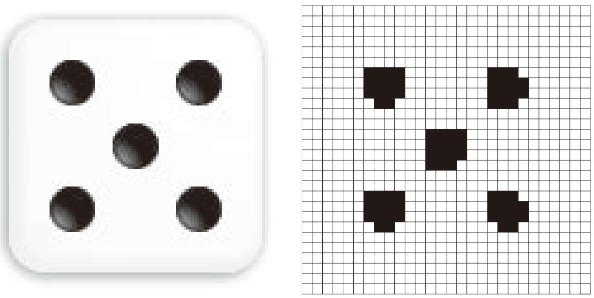


图 6.4　一个骰子的照片和海报化后的图片

- **深度优先算法**

深度优先遍历从顶点 u 开始，并且只能遍历从 u 开始可以抵达的所有顶点。所以，连通分量一定包含 u。我们以当前分量的编号作为访问标记。

```
def dfs_grid(grid, i, j, mark, free):
    grid[i][j] = mark
    height = len(grid)
    width = len(grid[0])
    for ni, nj in[(i + 1, j), (i, j + 1),                 # 四个相邻节点
                  (i - 1, j), (i, j - 1)]:
        if 0 <= ni < height and 0 <= nj < width:
            if grid[ni][nj] == free:
                dfs_grid(grid, ni, nj, mark, free)
```

① 注意，每个连通分量中的元素都用其编号来填充。——译者注

只要有连通分量，我们就一直执行深度优先遍历。横向和纵向遍历网格，一旦遇到一个包含 # 号字符的格子，我们就知道遇到了一个连通分量。然后，从这个格子开始深度优先遍历，确定连通分量的所有组成元素。

```
def nb_connected_components_grid(grid, free='#'):
    nb_components = 0
    height = len(grid)
    width = len(grid[0])
    for i in range(height):
        for j in range(width):
            if grid[i][j] == free:
                nb_components += 1
                dfs_grid(grid, i, j, str(nb_components), free)
    return nb_components
```

每个包含 # 字符的格子仅会被访问一次，所以算法的复杂度是 $O(|V|)$，也就是说，这个算法与顶点数量成线性关系。

- **使用并查集结构的算法**

这个图是无向图，所以“u 和 v 之间存在一条路径”和“v 和 u 之间存在一条路径”成等价关系。因此，连通分量就是这一关系的等价类型。因此，并查集是一个非常适于展示问题的数据结构（见 1.5.5 节）。

- **复杂度**

并查集结构算法的复杂度比深度优先算法略差。然而，假如要处理的图的边数会变动，而且需要随时知道连通分量的数量，那么并查集结构算法就十分必要。

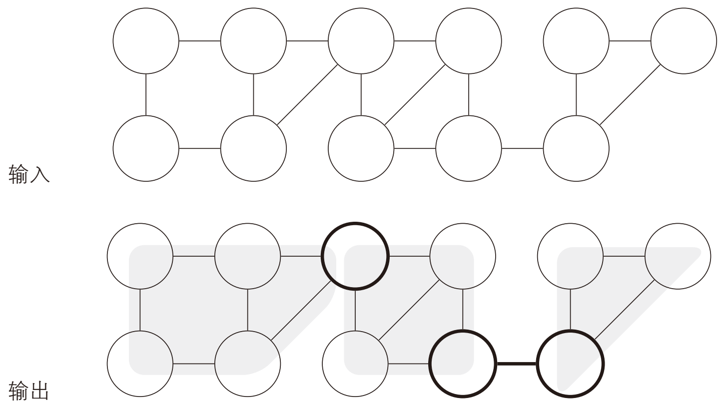
```
def nb_connected_components(graph):
    n = len(graph)
    uf = UnionFind(n)
    nb_components = n
    for node in range(n):
        for neighbor in graph[node]:
            if uf.union(node, neighbor):
                nb_components -= 1
    return nb_components
```

- **对一个图断开的应用**

假设一个图的边会随着时间逐渐消失，也就是说，这个图是一个随时间 i 前进而消失的边的序列 $e_1,\cdots, e_{|E|}$，比如边 e_i 会在 i 时刻消失。我们希望找到一个时间点，从这一刻开始，图不再是连通的。

我们在时间 $t = |E|$（包括 $|V|$ 个连通分量）从无边图开始。在每个步骤中，我们添加一条边并观察连通分量的数量是否变化。当连通分量的数量变成 1 时，我们知道图已经变成定义上的连通图，而且 $t+1$ 就是我们需要找的值。

6.7 双连通分量



- **应用**

给定一个图，图中每个顶点和每条边被破坏掉时都有一个成本值，我们希望找到唯一一个顶点或者唯一一条边，以便在把图变得不连通时成本最低。注意，这个问题和寻找一个边数最小的**集合**来断开图的问题有区别，后者是本书 9.8 节要介绍的最小切割问题。

- **定义：这是一个无向连通图。**

— **断开连接的顶点**，又称**衔接点**，被删除后就使图不再连通。

— **断开连接的边**，又称**桥**，被删除后就使图不再连通（图 6.7）。

— **双连通分量**是一个包含最大数量边的集合，对于一个（被限制在该边集合和所有邻接顶点范围内的）图来说，既不存在一条断开连接的边，也不存在一个断开连接的顶点。

— 不是桥的边，分布于双连通分量之间。

一个双连通分量 S 还有以下特性：对于所有顶点对 $(s, t)\in S$，一定存在从 s 到 t 且通过不同顶点的两条不同路径。注意，双连通分量是由一部分边定义的，而不是由一部分顶点定义的。其实，一个顶点可以属于多个双连通分量，如图 6.6 中的顶点 5。

对于一个给定的无向图，我们要把它拆分成多个双连通分量。

复杂度：使用深度优先算法的复杂度是线性的（见参考文献 [14]）。

- **细致的深度优先遍历**

在上文中，我们描述了图的深度优先算法。现在，要在顶点和边上增加附加信息。首先，顶点按照被处理的顺序编号。数组 dfs_num 保存了这个信息。

为每条边都生成两条弧，一个无向图就可以表示为有向图。深度优先算法遍历了图的所有边，我们用下面描述方式区分这些边（图 6.5）。

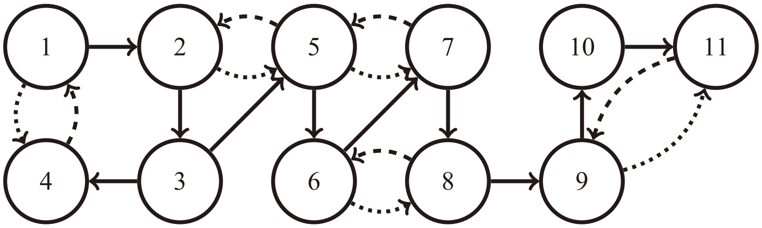


图 6.5 顶点按照被处理的顺序编号，保存在 dfs_num 中。实线表示连接弧，虚线表示回到顶点的弧，点虚线表示离开顶点的弧。对于每个连接弧 (u, v) 都存在着一个反向连接 (v, u)，为阅读方便，这个反向连接没有在图上标出

一条弧 (u, v) 可呈现以下形式。

— 连接弧：如果在处理 u 的时候，v 被第一次遇到。连接弧在遍历图的过程中形成了覆盖树，又称作 DFS 树。

— 反向连接弧：如果 (v, u) 是一条连接弧。

— 返回弧：如果 v 已经被遇到，且它在 DFS 树中是 u 的祖先。

— 离开弧：如果 v 已经被遇到，且它在 DFS 树中是 u 的后代。

在有向图的一次深度优先遍历中，还额外存在一类弧——**跨越弧**，它通向一个已经遇到的顶点，但该顶点既不是祖先顶点也不是后代顶点。我们在本节中不考虑无向图，因此可以忽略这类弧。

- **确定弧的类型**

通过比较 dfs_num 两端的值，很容易确定弧的类型（图 6.6）。具体来讲，对于每个顶点 v，除了 dfs_num[v]，我们还要确定算法的关键值之一 dfs_low[v]。它被定义为，当 w 是 v 的后代时，所有返回弧 (w, u) 的 dfs_num[u] 最小值。因此，这个最小值可从顶点 u 取到。通过一个（可以为空的）连接弧序列，并经过一条返回弧后从 v 可以抵达顶点 u。如果没有这种顶点 u，我们定义 dfs_num[u] = ∞。

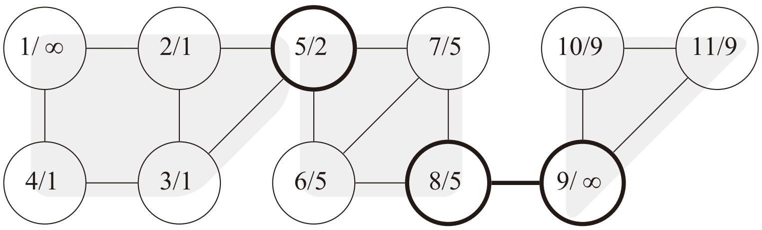


图 6.6 所有顶点被标注了 dfs_num 和 dfs_low，加粗的顶点和边是不连通的顶点和边

关键测试：上述这个值被用来确定顶点和不连通边。

1. 一个顶点 u 是 DFS 树的根节点，当且仅当它在树中拥有至少两个子节点时，它是不连通节

点。每个子节点 v 都满足 dfs_low[v] ⩾ dfs_num[u]。

2. 一个顶点 u 不是 DFS 树的跟节点，当且仅当它在树中拥有至少一个子节点 v，且满足 dfs_low[v] ⩾ dfs_num[u] 时，它是不连通节点。

3. 一条边 (u, v)（交换了 u 和旁边的 v），当且仅当 (u, v) 是一条连接弧且满足 dfs_low[u] ⩾ dfs_num[v] 时，它是一条不连通边。

为了确定双连通分量，只需在开始一个新双连通分量的时候应用上述定义即可（图 6.7）。

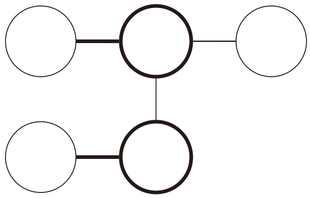


图 6.7 两个不连通节点中间的一条边不总是一条不连通边，一条不连通边的两端也不总是不连通节点

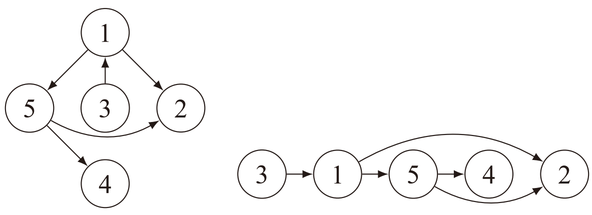
- **实现细节**

father 数组包含了 DFS 树中每个顶点的前驱，而且也可以确定 DFS 树的跟节点。对于每个顶点 u，我们在 critical_childs[u] 中记录子节点 v 在树中的数量，且 dfs_low[v] ⩾ dfs_num[u]。在确定每个从 v 出发的返回弧时，dfs_low[v] 的值被更新。在处理的最后，这个值被传播向 DFS 树中的父顶点。

```
# pour faciliter la lecture les variables sont sans préfixe dfs_
def cut_nodes_edges(graph):
    n = len(graph)
    time = 0
    num = [None] * n
    low = [n] * n
    father = [None] * n              # father[v] = f 的父节点，如果它是跟节点则是 none
    critical_childs = [0] * n        # c_childs[u] = nb fils v tq low[v] >= num[u]
    times_seen = [-1] * n
    for start in range(n):
        if times_seen[start] == -1:           # 初始化 DFS 遍历
            times_seen[start] = 0
            to_visit = [start]
            while to_visit:
                node = to_visit[-1]
                if times_seen[node] == 0:     # 开始处理
                    num[node] = time
                    time += 1
                    low[node] = float('inf')
                children = graph[node]
                if times_seen[node] == len(children):             # 结束处理
```

```
                    to_visit.pop()
                    up = father[node]                       # 把下层传播到父节点
                    if up is not None:
                        low[up] = min(low[up], low[node])
                        if low[node] >= num[up]:
                            critical_childs[up] += 1
                    else:
                        child = children[times_seen[node]]  # 下一条弧
                        times_seen[node] += 1
                        if times_seen[child] == -1:         # 还没访问过
                            father[child] = node            # 连接弧
                            times_seen[child] = 0
                            to_visit.append(child)          # （上方）返回弧
                        elif num[child] < num[node] and father[node] != child:
                            low[node] = min(low[node], num[child])
    cut_edges = []
    cut_nodes = []                                          # 输出结果
    for node in range(n):
        if father[node] == None:                            # 特征
            if critical_childs[node] >= 2:
                cut_nodes.append(node)
        else:                                               # 内部节点
            if critical_childs[node] >= 1:
                cut_nodes.append(node)
            if low[node] >= num[node]:
                cut_edges.append((father[node], node))
    return cut_nodes, cut_edges
```

6.8　拓扑排序



- **定义**

给定一个有向图 G(V, A)，我们希望把顶点按照等级 r 排序，使得对于每条弧 (u, v)，都有 $r(u) < r(v)$。

- **应用**

图可以表示一系列任务，其中从 u 到 v 的弧（$u \to v$）表示了 u 和 v 之间的依赖关系，即“一定要在 v 之前执行 u”。我们关心的是满足依赖关系的任务执行顺序。

首先有几点注意事项。

— 同一个图存在多种拓扑排序方式。比如，如果序列 s 是 G_1 的拓扑排序，而 t 是 G_2 的拓扑排序，那么 st 和 ts 都是 G_1 和 G_2 的并集组成的图的拓扑排序。

— 一个包含环的图上不能接纳拓扑排序：环的每个顶点都需要在其他顶点前被处理。

— 一个不包含环的图至少能接纳一种拓扑排序，详见下面的分析。

复杂度：输入数据大小决定的线性复杂度。

- **使用深度优先遍历的算法**

如果只在处理完一个顶点的所有相邻节点之后才处理顶点，我们就能得到一个反向拓扑排序。因此，如果 $u \to v$ 是一个依赖，那么：

— 要么 v 在 u 之前被遍历，此时 v 已被处理过（否则，这意味着从 v 可以抵达 u，那么图中包含一个环），而且逆序的拓扑顺序被满足；

— 要么是从 u 开始遍历到 v，此时 u 会在 v 之后被处理，逆序拓扑排序再次被满足。

前面介绍过的深度优先遍历在此处适用，因为它不能确定顶点被处理的结束日期。

以下实现方式使用数组 seen，数组用 -1 值来表示每个未被遇到的顶点；否则，这个值指出的是已被遍历顶点的直接后代数量。当这个计数器与某个节点的子节点数量一致时，对该节点的处理就结束了，然后它就会被添入序列 order。该序列保存着一个反向拓扑排序，必须在算法结尾处把该排序逆转。

```
def topological_order_dfs(graph):
    n = len(graph)
    order = []
    times_seen = [-1] * n
    for start in range(n):
        if times_seen[start] == -1:
            times_seen[start] = 0
            to_visit = [start]
            while to_visit:
                node = to_visit[-1]
                children = graph[node]
                if times_seen[node] == len(children):
                    to_visit.pop()
                    order.append(node)
                else:
                    child = children[times_seen[node]]
                    times_seen[node] += 1
                    if times_seen[child] == -1:
                        times_seen[child] = 0
                        to_visit.append(child)
    return order[::-1]
```

- **贪婪算法**

一个替代方案基于顶点的输入度。设想一个无环图。直观可知，我们首先把所有无前驱节点的节点强制加入结果序列，然后将其从图中删掉，再把图中无前驱节点的新节点加入结果序列，以此类推。这个过程最终会结束，因为一个无环图总存在一个无前驱节点的顶点，而且删除一个节点仍能保持图中没有环。

```
def topological_order(graph):
    V = range(len(graph))
    indeg = [0 for _ in V]
    for node in V:                                      # 确定输入度
        for neighbor in graph[node]:
            indeg[neighbor] += 1
    Q = [node for node in V if indeg[node] == 0]
    order = []
    while Q:
        node = Q.pop()                                  # 没有进入弧的顶点
        order.append(node)
        for neighbor in graph[node]:
            indeg[neighbor] -= 1
            if indeg[neighbor] == 0:
                Q.append(neighbor)
    return order
```

- **应用**

给定一个无环图和两个顶点 s 和 t，我们希望计算从 s 到 t 的路径数量，或者当弧上有权重时，找到最长[①]的一条路径。一个线性时间复杂度的算法使用了拓扑排序，并能按此顺序在节点上应用动态规划。

比如，动态规划 $P[s] = 0$，$P[v] = 1+\max_u P[u]$ 计算了从 s 到 t 的最长路径，其中，最长值的计算基于所有进入 v 节点的弧 (u, v)。

6.9　强连通分量

- **定义**

对于有向图的一部分 $A \subset U$，当 A 中所有顶点对 (u, v) 都包含着 A 内一条连接从 u 到 v 的路径时，A 就被称为强连通分量。注意，在这种情况下同样存在一条路径连接着从 v 到 u（图 6.8）。

---

① 即权重最大。——译者注

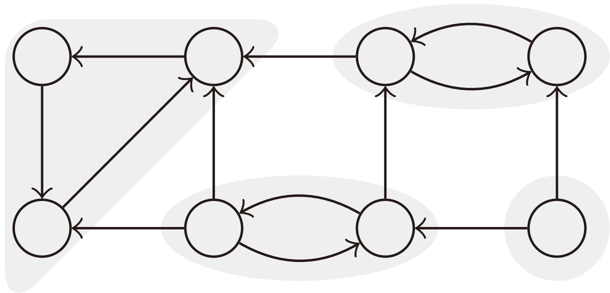


图 6.8　一个图的一部分是强连通分量

- **关键测试**

分量图是所有强连通分量收缩成超顶点后的结果。分量图是无环的，因为每个有向图的环都包含在唯一一个强连通分量中。

复杂度：使用以下算法能得到线性复杂度。

- **Tarjan 算法：**

Tarjan 算法（见参考文献 [27]）只执行一次深度遍历，并把所有顶点按照处理的时间顺序编号。算法同样会把遍历中遇到的顶点放入 waiting 栈，直到这些顶点能被分组到某个强连通分量中。当检测到一个分量时，就把它从 waiting 栈中删掉。这意味着，所有已被发现的分量的进入弧都会被忽略掉。

一次深度遍历会通过第一个顶点 v_0 进入一个分量 C，然后遍历分量中所有的顶点，甚至会遍历所有可以从 C 到达的顶点。当顶点 v_0 在 DFS 树中的所有后代节点都被处理完毕时，对顶点 v_0 的处理结束。从结构上看，显然在 v_0 处理结束的时候，waiting 栈在 v_0 之上，包含着所有 C 的顶点。我们把 waiting 栈称为 C 的**代表点**。顶点按照处理顺序编号，因此 C 的代表点编号最小。问题难在如何找到一个分量的代表点。

为此，算法为每个顶点 v 计算了两个值：一个是每个顶点被处理时被赋予的深度遍历顺序编号 dfs_num[v]；另一个是 dfs_min[v]，即所有尚未分入一个强连通分量的顶点 u 的 dfs_num[u] 最小值。这些顶点仍在等待分组。从顶点 v 出发，通过一系列连接弧就能达到这些顶点，而这些连接弧之后很可能跟着唯一一个返回弧，返回到 v。因此，dfs_min[v] 值也被定义为所有离开弧 (v, u) 的最小值。

$$\text{dfs_min[v]} := \min_u \begin{cases} dfs_num[v] & \\ dfs_\min[u] & \text{如果 } (v,u) \text{ 是 一条连接弧} \\ dfs_num[u] & \text{如果 } (v,u) \text{ 是一条返回弧} \end{cases}$$

注意，这个值与 6.7 节中描述的 dfs_low[v] 不同，后者不存在等待分组的顶点概念，而且 dfs_low[v] 可以取值∞。

假设一个没有离开弧的分量 C 和一个顶点 $v \in C$。同时假设 A 为根节点是 v 的 DFS 子树，而且在 v 之前存在一条离开 A 并指向顶点 u 的返回弧。由于 C 没有离开弧，u 是 C 的一部分且 v 不是 C 的代表点。在这种情况下，我们有 dfs_min[v] < dfs_num[v]。如果不存在离开 A 的返回弧，那么 A 包含一个强连通分量。由于 A 在 C 之内，这棵树就覆盖了 C，而 v 就是 C 的代表点。在 dfs_min[v]==dfs_num[v] 时，就可以看到这种情况。

- **实现细节**

在维护 waiting 栈的同时，算法维护一个布尔型数组 waits，用于在常数时间内检测顶点是否已经入栈。因此，通过把相关进入弧设置为 `False`，很容易就能把一个顶点从图中删除掉[①]。

算法返回其中一个数组，该数组包含每个分量的顶点。注意，这些分量是通过反向拓扑顺序确定的。我们后面求解一个 2-SAT 方程时，会用到这个算法。

```
def tarjan_recursif(graph):
    global sccp, waiting, dfs_time, dfs_num
    sccp = []
    waiting = []
    waits = [False] * len(graph)
    dfs_time = 0
    dfs_num = [None] * len(graph)

    def dfs(node):
        global sccp, waiting, dfs_time, dfs_num
        waiting.append(node)                # 新的等待顶点
        waits[node] = True
        dfs_num[node] = dfs_time            # 标注顶点已经被访问过
        dfs_time += 1
        dfs_min = dfs_num[node]             # 计算 dfs_min
        for neighbor in graph[node]:
            if dfs_num[neighbor] == None:
                dfs_min = min(dfs_min, dfs(neighbor))
            elif waits[neighbor] and dfs_min > dfs_num[neighbor]:
                dfs_min = dfs_num[neighbor]
        if dfs_min == dfs_num[node]:        # 一个分量的代表点
            sccp.append([])                 # 新建分量
            while True:                     # 把等待顶点加入分量
                u = waiting.pop()
                waits[u] = False
                sccp[-1].append(u)
                if u == node:               # 直到代表点
                    break
        return dfs_min

    for node in range(len(graph)):
        if dfs_num[node] == None:
            dfs(node)
    return sccp
```

① 也就是说，通过断开弧来断开顶点之间的连接，从而把一个顶点从一个分量中切掉。——译者注

- **迭代版本**

比如在处理 100 000 个顶点的大图时，需要使用算法的迭代版本。这里计数器 `times_seen` 能标注顶点是否被遇到，同时记录已被计算过的相邻节点数量。

```
def tarjan(graph):
    n = len(graph)
    dfs_num = [None] * n
    dfs_min = [n] * n
    waiting = []
    waits = [False] * n              # 常量： waits[v] 表示 v 是否在等待处理
    sccp = []                        # 已经确定的分量数组
    dfs_time = 0
    times_seen = [-1] * n
    for start in range(n):
        if times_seen[start] == -1:                        # 遍历初始化
            times_seen[start] = 0
            to_visit = [start]
            while to_visit :
                node = to_visit[-1]                        # 顶点的栈
                if times_seen[node] == 0:                  # 开始处理
                    dfs_num[node] = dfs_time
                    dfs_min[node] = dfs_time
                    dfs_time += 1
                    waiting.append(node)
                    waits[node] = True
                children = graph[node]
                if times_seen[node] == len(children):      # 结束处理
                    to_visit.pop()                         # 出栈
                    dfs_min[node] = dfs_num[node]          # 计算 dfs_min
                    for child in children:
                        if waits[child] and dfs_min[child] < dfs_min[node]:
                            dfs_min[node] = dfs_min[child]
                    if dfs_min[node] == dfs_num[node]:     # 代表点
                        component = []                     # 新建分量
                        while True:                        # 新增顶点
                            u = waiting.pop()
                            waits[u] = False
                            component.append(u)
                            if u == node:                  # 直到代表点
                                break
                        sccp.append(component)
                else:
                    child = children[times_seen[node]]
                    times_seen[node] += 1
                    if times_seen[child] == -1:            # 还没有访问过
                        times_seen[child] = 0
                        to_visit.append(child)
    return sccp
```

- **Kosaraju 算法**

Kosaraju 提出了一种不同的算法（见参考文献 [20]），复杂度同样是线性的。在现实中，Kosaraju 算法的复杂度与 Tarjan 算法接近，但更容易理解。

算法的核心在于首先执行一次深度优先遍历，然后在把所有弧反向后的图上执行第二次深度优先遍历。通常，公式 $A^T := \{(v, u)(u, v) \in A\}$ 记录了所有弧反向之后的结果。算法分为以下两个步骤。

1. 对 G(V, A) 执行深度优先遍历，使用 $f[v]$ 来记录处理顶点 v 的结束时间。
2. 对 $G(V, A^T)$ 执行深度优先遍历，以 $f[v]$ 降序排列后的根节点 v 作为遍历源点。

在第二次遍历中，每个遇到的树形结构都是一个强连通分量。

验证算法的基本思路是，如果把每个强连通分量 C 与整数 $F(C) := \max_{u \in C} f_u$ 相关联，那么 F 能通过处理 $G(V, A^T)$ 的强连通分量，得到一个拓扑排序。因此在第二次遍历中，每个树形结构都留在一个分量内部，因为只有离开弧会指向已访问过的分量。

- **实现细节**

数组 sccp（strongly connected component）包含了所有强连通分量的列表。

```
def kosaraju_dfs(graph, nodes, order, sccp):
    times_seen = [-1] * len(graph)
    for start in nodes :
        if times_seen[start] == -1:                     # 初始化深度优先遍历
            to_visit = [start]
            times_seen[start] = 0
            sccp.append([start])
            while to_visit:
                node = to_visit[-1]
                children = graph[node]
                if times_seen[node] == len(children):   # 结束处理
                    to_visit.pop()
                    order.append(node)
                else:
                    child = children[times_seen[node]]
                    times_seen[node] += 1
                    if times_seen[child] == -1:         # 新节点
                        times_seen[child] = 0
                        to_visit.append(child)
                        sccp[-1].append(child)

def reverse(graph):
    rev_graph = [[] for node in graph]
    for node in range(len(graph)):
        for neighbor in graph[node]:
            rev_graph[neighbor].append(node)
    return rev_graph

def kosaraju(graph):
```

```
    n = len(graph)
    order = []
    sccp = []
    kosaraju_dfs(graph, range(n), order, [])
    kosaraju_dfs(reverse(graph), order[::-1], [], sccp)
    return sccp[::-1]                # 使用拓扑逆序
```

6.10 可满足性

很多决策问题都可以采用“是否满足布尔方程”的思路来建模，这就是可满足性问题。

- **定义**

假设有 n 个布尔型变量。一个**命题**是一个变量或一个变量的逆值。一个**语句**是多个命题的“或组合”，也就是说，当至少有一个命题为真时，这个语句成立。一个**公式**是多个命题的“与组合”，也就是说，只在所有命题都为真时，这个公式成立。最终目的是弄清是否存在某个给变量赋值的方式，使方程成立。

当每个语句都最多包含两个命题时，一个方程就被定义为 2-SAT 级别。信息科学的一个基础问题就是，能否证明在线性时间内找到使一个 2-SAT 方程成立的结果，然而一般来说（以 3-SAT 方程为例），我们在最坏情况下不知道在多项式时间内解决问题的算法。

复杂度：线性。

- **使用有向图建模**

两个命题的逻辑或（$x \vee y$）等价于 $\overline{x} \Rightarrow y$，甚至 $\overline{y} \Rightarrow x$。我们把这个等价图与一个 2-SAT 方程相关，其中顶点是命题，弧与语句等价。图 6.9 展现了严格的对称性。

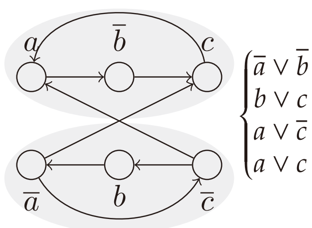


图 6.9　2-SAT 实例图。图中有两个强连通分量，下方分量指向上方分量。因此，把下方所有命题赋值为 false，并把上方所有命题赋值为 true，就能使方程成立

- **关键测试**

很容易证明，如果在等价图中存在一个变量 x，并存在一条从 x 到 $\overline{x}$ 的路径和一条 $\overline{x}$ 到 x 的路径，那么 2-SAT 方程不成立。出人意料的是，这个命题的逆命题同样成立。

由于关联有传递性，因为当方程能推导出 $x \Rightarrow \bar{x} \Rightarrow x$ 时，方程不可能成立。因为每次把 x 赋值为 false 或 true 时，都会导致矛盾的结果 false $\Rightarrow$ true。对于逆命题，假设所有变量的逆命题都在另一个强连通分量中。由于有向图是对称的，我们可以考虑，每一个强连通分量中都包含着另一个分量所有命题的逆命题。另外，同一个强连通分量中的所有命题必须拥有相同的布尔值。利用这个图，只需把一个分量中所有不包括离开弧的命题赋值为 true，就可能找到一个使方程成立的赋值。这个赋值一定存在，因为分量组成的图是无环的。然后，把相对分量赋值为 false，再把这两个分量切开，重新开始。

为了更有效率，上述找到强连通分量的算法是按拓扑逆序将强连通分量排序的。只需按照这个顺序遍历分量，把每个没有值的分量赋值为 true，然后把相对分量赋值为 false。

- **实现细节**

我们用整形数对命题进行编码，如用 +1, ⋯, +n 来记录 n 个命题变量，用 -1, ⋯, -n 来记录它们的逆命题。一个语句通过一个整数对来编码，一个方程通过一个语句的数组来编码。每个命题都与等价有向图 $2n$ 个节点中的一个节点相关。这些节点从 0 到 $2n$–1 编号，其中 $2i$ 表示变量 x_{i+1}，而 $2i$+1 表示 $\bar{x}_{i+1}$。相关代码如下。

```
def _vertex(lit):                                   # 用顶点表示给定命题
    if lit > 0:
        return 2 * (lit - 1)
    else:
        return 2 * (-lit - 1) + 1

def two_sat(formula):
    #                                   -- n 是变量数量
    n = max(abs(clause[p]) for p in(0, 1) for clause in formula)
    graph = [[] for node in range(2 * n)]
    for x, y in formula:                            # x 或 y
        graph[_vertex(-x)].append(_vertex(y))       # -x => y
        graph[_vertex(-y)].append(_vertex(x))       # -y => x
    sccp = tarjan(graph)
    comp_id = [None] * (2 * n)                      # 分量的每个节点的 id
    affectations = [None] * (2 * n)
    for component in sccp:
        rep = min(component)                        # 分量的代表点
        for vtx in component:
            comp_id[vtx] = rep
            if affectations[vtx] == None:
                affectations[vtx] = True
                affectations[vtx ^ 1] = False       # 逆命题
    for i in range(n):
        if comp_id[2 * i] == comp_id[2 * i + 1]:
            return None                             # 方程不成立
    return affectations[::2]
```

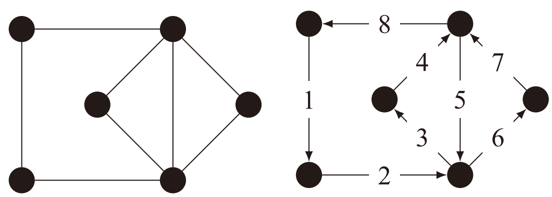
第 7 章　图中的环

许多经典问题都与图中的环相关，比如地理移动问题或一个依赖图中的异常问题。最简单的问题是确定环的存在性、负权重环的存在性，以及总权重最小的环或平均权重最小的环的存在性。

其他问题旨在遍历整个图，计算仅经过每条边一次的路径（欧拉路径），或者当不可能实现该目标时，计算至少经过每条边一次的路径（中国邮差问题）。这些问题都有多项式时间复杂度，因为确定一个能准确通过所有顶点一次的环（哈密顿环）是 NP 复杂问题。

搜索环的算法		
定位环	$O(\|V\|+\|E\|)$	深度优先遍历
总权重最小的环	$O(\|V\|\cdot\|E\|)$	Bellman-Ford 算法（最短路径算法）
平均权重最小的环	$O(\|V\|\cdot\|E\|)$	Karp 算法
最优比率环	$O(\|V\|\cdot\|E\|\cdot\log\sum t)$	二分查找
欧拉路径	$O(\|V\|+\|E\|)$	穷举算法
中国邮差环	$O(\sqrt{\|V\|}\cdot\|E\|)$	最小权重完美连接
旅行商问题	$O(\|V\|2^{\|V\|})$	动态规划

7.1　欧拉路径



输入　　　　输出

- **应用**

你在加里宁格勒（旧称柯尼斯堡）旅游，能否找到一条游览路径，要求仅通过城中所有桥一次，而且还能回到出发点？这就是欧拉在 1736 年研究的问题情景[①]。

- **定义**

给定一个连通图 G(V, E)，图的每个顶点都是偶数价的[②]。我们要在图中找到精确经过每条边一次的环。有向图和强连通分量也有同样的问题，这时会要求离开顶点的价与进入顶点的价一致。

- **线性时间复杂度的算法**

当且仅当一个图的所有顶点都是偶数价的时候，它才包含欧拉路径。这一点已在 1736 年被欧拉证明。同样，对于一个有向图，当且仅当它是连通图，且其所有顶点的离开价等于进入价时，它才包含欧拉路径。怎样才能找到这样一个环？ 1873 年，Hierholzer 提出了以下算法。随意找一个顶点 v，从 v 出发把所有经过的边都标记为不可通过。边的选择也可以是随机的。这样走一定能返回顶点 v，因为路线末端只能是仅有奇数条可通过边的邻接顶点。如此得到的环 C 仅覆盖一部分图。在这种情况下，由于图是连通的，因而一定存在一个顶点 $v' \in C$，它有可通过的边。我们从 v' 开始新路程，再次得到新的环 C。不断重复上述过程，直至找到唯一一条欧拉路径。

为了让算法拥有线性时间复杂度，对顶点的搜索一定要足够高效。因此，我们把环 C 切成 P 和 Q 两部分。P 中的顶点没有邻接的可通过边。只要 Q 非空，我们把 Q 开端的顶点 v 删除并加入 P 的尾部——就像环在向前滚动一样。然后，我们试着在这个插入顶点 v 的地方加入一个通过 v 的环。为此，当 v 有一条邻接可通过边时，我们仅需遍历、寻找一个从 v 出发且回到 v 的环 R 即可。接下来，我们把环 R 加入 Q 的开端。由于每条边仅被考虑一次，于是算法有了线性时间复杂度。

- **实现细节**

我们从有向图的算法实现开始。为了简化数据控制，我们使用以数组编码的栈来代表 P、Q、R。当前队列由栈 P 表示，其后是 R，再后是 Q 的镜像数组。为了快速找到一条离开某一顶点的可通过边，我们在计数器 next[node] 中保存离开顶点 node 和已经过的弧的数量。当我们通过弧达到 node 的第 i 个邻点，且 i = next[node] 时，只需增加这个计数器的值。

```
def eulerian_tour_directed(graph):
    P = []
    Q = [0]
    R = []
    next = [0] * len(graph)
    while Q:
        node = Q.pop()
```

① 加里宁格勒是一座俄罗斯城市，与波兰和立陶宛接壤，是俄罗斯的一块飞地。城中有七座桥将普列戈利亚河中两个岛，以及岛与河岸连接起来，因此这个问题也称作“欧拉七桥”问题，奠定了现代图论和拓扑学的基础。——译者注

② 连接顶点的边或弧的数量是偶数。——译者注

```
        P.append(node)
        while next[node] < len(graph[node]):
            neighbor = graph[node][next[node]]
            next[node] += 1
            R.append(neighbor)
            node = neighbor
        while R:
            Q.append(R.pop())
    return P
```

该算法的无向图变种颇为巧妙。一旦弧 (u, v) 被通过，需要把弧 (v, u) 标注为不可通过。我们在数组 seen[v] 中保存 v 的相邻顶点 u 的集合，使得弧 (v, u) 不可被通过。为了提高效率，v 在通过弧 (u, v) 的时候不会被加入 seen[u]，因为此时计数器 next[u] 已经增加，这条弧不会再被考虑了。

```
def eulerian_tour_undirected(graph):
    P = []
    Q = [0]
    R = []
    next = [0] * len(graph)
    seen = [set() for _ in graph]
    while Q:
        node = Q.pop()
        P.append(node)
        while next[node] < len(graph[node]):
            neighbor = graph[node][next[node]]
            next[node] += 1
            if neighbor not in seen[node]:
                seen[neighbor].add(node)
                R.append(neighbor)
                node = neighbor
        while R:
            Q.append(R.pop())
    return P
```

- **欧拉路径的变种**

如果一个图是连通图，而且所有顶点的价都是偶数——除了两个顶点 u 和 v 的价是奇数，那么存在一条路径仅一次通过所有的边。这条路径从 u 开始到 v 结束。只需从 u 开始走，通过上述算法就能找到这条路径。为了证明这一点，只需临时添加边 (u, v)，并寻找一个欧拉环即可。

7.2 中国邮差问题

- **应用**

1962 年，中国数学家管梅谷做起了邮递员工作。他提出了在图中找到至少遍历每条边一次且总距离最短的路径问题。这正是在城市各街道间分发邮件的邮递员必须解决的问题。

- **定义**

指定一个无向连通图 G(V, E)。我们的最终目的是在这张图中找到一条能够至少经过每条边一次的环。当所有顶点的价都是偶数时，解决方法是找到一条欧拉路径，上一节中已经介绍过解法。

复杂度：$O(n^3)$（见参考文献 [5]）。

- **算法**

算法的核心在于处理一张伪图，即一个允许两个相同顶点间有多条边的图。思路是添加边，使图**欧拉化**，并形成一条欧拉环路。新增的边必须把奇数价的顶点连接起来，使图欧拉化。我们希望尽可能少地添加边。这些额外添加的边会形成一个路径集合，使得奇数价顶点变成偶数价顶点。

因此，问题的核心就变成在一个完整的图中，使用所有奇数价顶点集合 V 计算一个完美分割，使得在图 G 中，一条边 (u, v) 的权重等于 u 和 v 之间的距离。

使用 Floyd-Warshall 算法计算所有距离需要 $O(n^3)$ 复杂度。此外，以 Gabow 算法计算最小权重的完美分割可以在时间 $O(n^3)$ 内完成（见参考文献 [9]），但对本书寻求高效算法的主旨而言，该算法过于复杂了。

7.3　最小长度上的比率权重环：Karp 算法

一个经典问题是在一个图中找到负环。一个应用将在后面章节中作为练习给出。给定 n 种货币及其兑换汇率，如何通过交易货币来挣钱？在本节中，我们只讨论一个解法更优雅的问题。

- **定义**

给定一个有权重的有向图，目的是找到一个环 C，使得环经过的弧的权重平均值 $\frac{\sum_{e\in C} w(e)}{|C|}$ 最小。

- **应用**

假设用一个图的顶点和弧分别给一个系统的状态和变化建模，每个变化（一条弧）都用所需消耗的资源量来标注权重。在每个时间节点上，系统都处于一个特殊状态，而且要通过离开弧来进化到下一个状态。我们的目的是让长期资源消耗最小化，而最优方案是一个最小平均权重环。

- **复杂度为 $O(|V|\cdot|E|)$ 的 Karp 算法（见参考文献 [16]）**

算法假设存在一个从任意顶点都能到达的源点。必要时，可以添加这样一个顶点到图中。

由于问题焦点是环中弧的平均权重，因而环本身的长度也同样重要[①]。因此，相对于简单计算出最短路径的方案，我们更希望能针对每个顶点 v 和每条弧长 $l = 0,\cdots, n$，确定一条从源点到 v 的最短

---

① 平均权重是用权重总和除以弧的数量。在权重相等的情况下，弧的数量增多则平均值降低。

——译者注

路径，该路径准确地由 l 条弧组成。这部分可通过动态规划来实现。以 $d[l][v]$ 来表示这条最短路径的权重（即从源点到 v 有 l 条弧）。起初，$d[0][v]$ 的值相对于源点是 0，对于其他所有顶点是；此后，该值对于每个 $l=1,\cdots,n$ 有：

$$d[l][v]=\min_u d[l-1][u]+w_{uv}$$

其中，当进入弧 (u,v) 的权重 w_{uv} 最小时，等式能得到最小值[①]。我们用 $d[v]=\min_{k\in N} d[k][v]$ 来记录从源点到 v 的距离。

- **关键测试**

对于一个最小平均权重环 C，其权重 λ 如下[②]：

$$\lambda=\min_v \max_{k=0,\cdots,n-1} \frac{d[n][v]-d[k][v]}{n-k} \tag{7.1}$$

为了证明这一点，只需从 $\lambda=0$ 开始一系列测试。我们必须证明以上公式的右边等于 0，这相当于证明：

$$\min_v \max_{k=0,\cdots,n-1} d[n][v]-d[k][v]=0$$

由于 $\lambda=0$，图中不包含负权重的环。对于所有顶点 v，一定存在一条从源点到顶点 v 的非环最短路径，即[③]

$$d[v] \max_{k=0,\cdots,n-1} d[k][v]$$

因此[④]

$$\max_{k=0,\cdots,n-1} d[n][v]-d[v]=d[n][v]-d[v]$$

对于所有满足 $d[n][v]\geqslant d[v]$ 的顶点 v，有[⑤]

$$\min_v d[n][v]-d[v]\geqslant 0$$

剩下要做的只是证明对于一个顶点 v，有等式 $d[n][v]=d[v]$。设一个环 C 的顶点 u，由于不存在负环，因而一定存在一条从源点到 u 的权重最小的简单路径 P。我们把环 C 的复本补全到 P，得到一

---

① 利用动态规划的思想，把解 $d[l][v]$ 拆分成 $d[l-1][u]$ 的最小权重和弧 (u,v) 的权重。——译者注

② 在最小平均权重环 C 的权重公式中，v 是一个从所有顶点都能到达的顶点。从某个顶点开始经过 n 条弧的距离减去经过 k 条弧的距离，除以 n 和 k 的差，就是 n 和 k 间的平均距离，变动 v、n、k，找到一个总体平均值最小的 λ 就是最小平均权重环的权重。——译者注

③ 源点到 v 的最短路径是经过 0 条弧到 n-1 条弧的所有路径中最短的一条。——译者注

④ 公式右边的 $d[n][v]$ 是从 v 出发经过 n 条弧的路径长度，减去从源点到 v 的距离，该值一定是所有路径中最长的一条，也就是说，走了最多 n 条弯路的路径。——译者注

⑤ 很明显，在从 v 出发的路径中，最短的一条是返回其本身的情况，即 $n=0$。其他情况只要经过任意一个弧都会超过它的长度，因此，经过任意多条弧的路径最小值减去 v 到源点的距离仍大于等于 0。——译者注

条路径 P′，该路径从源点到 u 且长度至少为 n。由于 C 的权重为空（ $\lambda=0$ ），P′ 仍是通向 u 的最小权重的路径。设 P″ 为 P′ 的前缀[①]，长度是 n; v 是 P″ 的最后一个顶点，P″ 也是从源点到达 v 的最短路径（图 7.1)。因此，对于这个顶点有 $d[n][v] = d[v]$，这就证明了在 $\lambda=0$ 的情况下，等式成立。

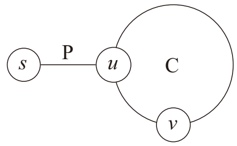


图 7.1　对 $\lambda=0$ 情况的证明。P 是从源点 s 到 u 的最小权重路径。添加一个权重为 0 的环 C，使得路径仍是最小权重路径，因此，在环中遇到的所有顶点即在最小权重路径中经过的顶点

对于 $\lambda \neq 0$ 的情况，我们来做一个有趣的测试。如果从每个弧的权重中去掉一个值 Δ ，λ 的值也只会减少 Δ 。如果用 d' 来记录图中被改动过的距离，会得到下列等式：

$$\frac{d'[n][v]-d'[k][v]}{n-k} = \frac{d[n][v]-n\Delta-(d'[k][v]-k\Delta)}{n-k}$$

$$= \frac{d[n][v]d[k][v]}{n-k} - \frac{n\Delta-k\Delta)}{n-k}$$

$$= \frac{d[n][v]d[k][v]}{n-k} - \Delta$$

这证明了等式 (7.1) 的右边同样会减少 Δ 。为 Δ 选择常数 λ ，最终同样会得到 $\lambda=0$ 的情况，这就证明了等式 (7.1)。

- **实现细节**

矩阵 `dist` 保存着上述距离列表。除了这个矩阵，还需要一个变量 `prec` 来表示一条最短路径的前驱顶点。在填充好这些矩阵后，我们需要找到顶点对 (v, k) 来优化表达式 (7.1)，然后提取出环。假如从源点出发找不到任何环，函数返回 `None` 值。

```
def min_mean_cycle(graph, weight, start=0):
    INF = float('inf')
    n = len(graph)                                   # 计算距离
    dist = [[INF] * n]
    prec = [[None] * n]
    dist[0][start] = 0
    for ell in range(1, n + 1):
        dist.append([INF] * n)
        prec.append([None] * n)
        for node in range(n):
            for neighbor in graph[node]:
```

① 达成 P' 要求的前一个步骤需要达成 P"，即动态规划中的问题分解思路。——译者注

```
            alt = dist[ell - 1][node] + weight[node][neighbor]
            if alt < dist[ell][neighbor]:
                dist[ell][neighbor] = alt
                prec[ell][neighbor] = node

#                         -- 确定最优值
valmin = INF
argmin = None
for node in range(n):
    valmax = -INF
    argmax = None
    for k in range(n):
        alt = (dist[n][node] - dist[k][node]) / float(n - k)
        # 总权重最小的环不除以 float(n - k)
        if alt >= valmax:           # 使用 >=，  寻找简单环
            valmax = alt
            argmax = k
    if argmax is not None and valmax < valmin:
        valmin = valmax
        argmin = (node, argmax)
#                             -- 提取环
if valmin == INF:           # -- 完全没有环
    return None
C = []
node, k = argmin
for l in range(n, k, -1):
    C.append(node)
    node = prec[l][node]
return C[::-1], valmin
```

7.4 单位时间成本最小比率环

- **定义**

一个有向图的每条弧上都有两个权重，分别是成本 c 和时间 t。时间为正值或空，而成本是随机的。算法的目的是找到一个环，使总成本和总时间的比值最小，问题又称作“不定线货船问题”。

- **应用**

一条商船的船长希望找到一条收益最大的航海路线。他手头有一张海图，完整覆盖了所有港口和每条港口间航线，每条弧 (u, v) 都被标注了从 u 出发到达 v 所需的时间，以及从 u 采购商品到 v 销售能够获得的利润。我们的任务是找 到一个环，在总时间内让收益达到最高。

- **使用二分查找的算法**

对于一个环 C，条件是当且仅当[①]

$$\frac{\sum_{a\in C}c(a)}{\sum_{a\in C}t(a)}\leqslant\delta\ ,\qquad \sum_{a\in C}c(a)\leqslant\sum_{a\in C}\delta t(a)\ ,\qquad \sum_{a\in C}c(a)-\delta t(a)\leqslant 0\ .$$

其目标值至少是 δ 。所以，想找到一个比值比 δ 更好的环，只需在图中找到一个弧的权重值是 $c(a)-\delta t(a)$ 的负环。这个测试可以被用在二分查找中，以得到指定精度的答案。当所有成本和时间权重值都是整数时，精度只需达到 $1/\sum_a t(a)$ 就可以精确解决这个问题。因此，算法的时间复杂度是 $O(\log(\sum t(a))\cdot|V|\cdot|A|)$ ，其中 V 是顶点的集合，A 是弧的集合。

由于刚开始没有最佳值 δ 的上界或下界，我们从 $\delta=1$ 开始测试。当测试结果为负值时，我们就把 δ 乘以 2，以便获得正值结果 δ' ；当测试结果为正值时，即得到最佳值的上、下界。当计算初始值是正值时，我们继续原有操作，但要除以 2 再继续[②]。

7.5　旅行推销员问题

- **定义**

给定一个图，弧上标注了权重，我们希望计算出一条从指定点出发的最短路径，使路径能准确经过每个顶点各一次。这样的路径称为“哈密顿路径”。

复杂度：使用动态规划情况下为 $O(|V|2^{|V|})$。

- **算法**

这种决策问题是一个 **NP** 完备问题，我们现在介绍一个当顶点数在 20 个左右时的可接受算法。为方便描述，假设顶点从 0 开始编号直到 n–1，且编号 n–1 的顶点是源点。对于每个集合 $S\subseteq\{0,1,\cdots,n-2\}$ ，我们用 $O[S][v]$ 来记录从源点出发，通过 S 中所有顶点并终止于顶点 v（$v\notin S$）的路径的最小权重。

对于基本情况，$O[\varnothing][v]$ 就是从编号 n-1 的顶点到顶点 v 的弧长。否则对于非空的 S，弧长为 $O[S][v]$，且当所有顶点 $u\in S$ 时，它是公式

$$O[S\backslash\{u\}][u]+w_{uv}$$

的最小值，其中 w_{uv} 是弧 (u, v) 的权重。

---

① 最左边的算式，一个环的路径上所有弧的成本之和除以经过所有弧的时间之和，可以理解为性价比。假如我们需要让这个值最大，即所有能找到的环的单位收益都小于等于这个值，那么反推就可以得到使它成立的条件，即最右边的算式。——译者注

② 二分查找的核心是先指定一个上界和下界，然后把上、下界一分为二，判断两部分中哪一部分符合测试要求，然后把符合要求的那一部分的上、下界作为新的上、下界继续查找。——译者注

第 8 章　最短路径

图论的一个经典问题是找到两个顶点——源点 s 和目标顶点 v 之间的最短路径。在成本不变的情况下，我们可以找到源点 s 和所有可能目标顶点 v' 之间的最短路径。因此，在本章介绍的算法中，我们对有向图中有唯一源点的普适问题更感兴趣。

一条路径的长度被定义为其所有弧的权重总和。从 s 到 v 的距离被定义为 s 和 v 之间最短路径长度。为方便表述，我们仅简单展示如何计算距离。为了获得一条满足要求的路径，在距离数组之外，只需维护一个前驱顶点数组。因此，对于一个顶点 u，如果 dist[v] 是从 s 到 v 的距离且 dist[v] = dist[u] + w[u][v]，那么在前驱顶点数组中保存 prec[v] = u。从前驱顶点回溯到源点 s，我们就可以用逆序法确定一条从源点到指定目标顶点的最短路径。

8.1　组合的属性

最短路径拥有组合属性，这是寻找最短路径的不同算法之间的关键差异。Bellman 称之为“最优化原则”，这也是动态规划问题的核心。让我们考虑一条从 s 到 v 的路径 P（也称 s-v 路径），它经过一个顶点 u（图 8.1）。因此，这是一条从 s 到 u 的路径 P_1 和一条从 u 到 v 的路径 P_2 的拼接。P 的长度是 P_1 和 P_2 的长度和。所以，如果 P 是从 s 到 v 的最短路径，那么 P_1 必定是从 s 到 u 的最短路径，而且 P_2 也必定是从 u 到 v 的最短路径。这个结论的证明很简单，假如我们能用一条更短的路径替换 P_1，那一定会得到一条比 P 更短的路径。

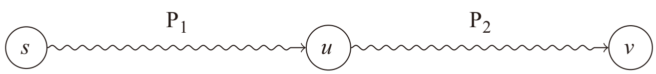


图 8.1　从 s 出发、经过 u 到 v 的最短路径，是由从 s 到 u 的最短路径和从 u 到 v 的最短路径组成的

- **黑色、白色、灰色顶点**

组合属性是 Dijkstra 算法及其变种的基础，用于弧上包含正值和空值权重的图。算法维护数组 dist 来保存从源点 s 到目标顶点 v 的距离；对于没有找到任何 s-v 路径的目标顶点 v，保存 $+\infty$。因此，图的顶点被分成三组（图 8.2）。黑色顶点是从源点出发的已知最短路径顶点，灰色顶点是黑色顶点的直接相邻顶点，白色顶点是还没有找到任何路径的顶点。

刚开始，只有源点 s 是黑色的，其 dist[s]=0。s 的直接相邻顶点都是灰色的，dist[v] 是弧 (s, v) 的权重。其他顶点都是白色的。然后，算法循环标注一个顶点的颜色为黑色或灰色，并把其相邻白色顶点标注为灰色，其他维持不变。最终，所有从源点可到达的顶点都会被标注为黑色，而其他顶点会是白色。

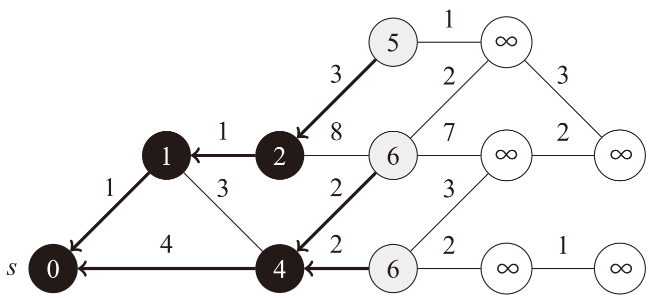


图 8.2　使用 Dijkstra 算法标注顶点颜色。每个顶点 v 都以从源点到其之间的距离 dist[v] 来标注，用 prec 表示的弧以粗体显示

- **关键测试**

哪个灰色顶点会被选中并标注为黑色？答案是令 dist[v] 最小的灰色顶点。为什么这一选择能奏效呢？让我们考虑一个随机的 s-v 路径 P。由于 s 是黑色的而 v 是灰色的，因而在这条路径上一定存在一个顶点 u 是灰色的。所以，P 可以拆分为一条 s-u 路径 P_1 和一条 u-v 路径 P_2；其中 P_1 仅包含黑色的中间顶点，除了 u。通过选择 v，且 dist[u] $\geqslant$ dist[v]，并假设所有弧的权重都是正值或空值，那么 P_2 的长度就是正值或空值。所以，P 的长度一定至少是 dist[v]。由于这个 P 是随机选择的，这就证明了最短路径 s-v 的长度是 dist[v]。因此，把 v 标注为黑色是有效的。为了维护状态不变的顶点，必须保证能从 v 出发并经过一条弧抵达每个顶点 v'；当 v' 不是灰色时，把 v' 标注为灰色，而 dist[v]+w[v][v'] 是找到 dist[v'] 的一个新候选方案。

最短路径算法		
没有权重	$O(\|E\|)$	广度优先遍历（BFS）
权重为 0 或 1	$O(\|E\|)$	使用双向队列的的 Dijkstra 算法
权重为正值或空值	$O(\|E\|\log\|V\|)$	Dijkstra 算法
随机权重	$O(\|V\|\cdot\|E\|)$	Bellman-Ford 算法
所有源点	$O(\|V\|^3)$	Floyd-Warshall 算法

- **灰色顶点的数据结构**

在每次遍历过程中，我们都要寻找一个灰色顶点 v 使得 dist[v] 最小，所以使用一个优先级序列来存储顶点 v 的候选者是合理的，dist 的值成了优先级的值。这就是 Dijkstra 算法选择的实现方式。因此，用最短距离来选择顶点可以在顶点数量的对数时间内完成。

如果图比较简单，我们可以用一个更简单的数据结构。比如，当所有弧的权重都在集合 {0, 1} 中时，只可能存在两种类型的灰色顶点，即距离为 d 的顶点和距离为 d+1 的顶点。因此，优先级队列可以采用更简单的双向队列来实现。队列包含了用优先级排序的灰色顶点列表，只需在常数时间内从队列左侧抽取一个顶点 v 并标注为黑色。v 的相邻顶点将根据其相关弧的权重是 0 还是 1 被添加到队列的左侧或右侧（图 8.2）。最终，所有队列的操作时间都是常数时间，相对于 Dijkstra 算法，这能节省一个对数因子的时间。

如果图还要更加简单，即所有弧的权重都相同——在某种程度上这就是无权重图，那么双向队列可以被简单队列来替换。请参阅 6.5 节介绍的广度优先算法。

8.2　权重为 0 或 1 的图

- **定义**

给定一个图，其所有弧的权重都是 0 或 1，同时给定一个源点 s，希望计算 s 到其他顶点的距离。

- **应用**

假设有一张 $N \times M$ 的矩形迷宫地图，迷宫里有障碍物。你希望在拆掉尽可能少的墙的情况下，找到走出迷宫的方法。这个迷宫可以被视为一个有向图，从一个格子到相邻格子的弧的权重要么是 0（通向一个空格），要么是 1（通向一个有障碍物的格子）。现在要尽可能少拆墙，找到从起点到出口的最短路径。

- **算法**

我们使用最短路径算法的通用结构。在任何情况下，图中所有顶点都被分成三组：黑色、灰色和白色。

算法维护一个双向队列，队列保存所有灰色顶点以及在插入时是灰色但会变成黑色的顶点。队列优先级的值是 x，所有黑色顶点 v 满足 dist[v] = x。直到某个特定位置，所有灰色顶点 v 满足 dist[v]=x，而后续顶点满足 dist[v] = x+1。

一旦这个队列非空，算法从队列头部提取顶点 v，其值 dist[v] 一定是最小的。如果这个顶点已经是黑色的，就不需要任何操作；否则，该顶点被标注为黑色。从现在开始，为了维护那些不变的顶点，需要把 v 的某个相邻顶点 v' 加入队列。对于 l = dist[v]+w[v][v']，如果 v' 已经是黑色或者 dist[v'] $\leqslant l$，不必把 v' 加入队列；否则 v' 被标注为灰色，dist[v'] 减小 l，且在 w[v][v'] = 0 的情况下，

v' 被加入到队列头部，或在 w[v][v'] = 1 的情况下，被加入队列尾部。

```
def dist01(graph, weight, source=0, target=None):
    n = len(graph)
    dist = [float('inf')] * n
    prec = [None] * n
    black = [False] * n
    dist[source] = 0
    gray = deque([source])
    while gray:
        node = gray.pop()
        black[node] = True
        if node == target:
            break
        for neighbor in graph[node]:
            ell = dist[node] + weight[node][neighbor]
            if black[neighbor] or dist[neighbor] <= ell:
                continue
            dist[neighbor] = ell
            prec[neighbor] = node
            if weight[node][neighbor] == 0:
                gray.append(neighbor)
            else:
                gray.appendleft(neighbor)
    return dist, prec
```

8.3　权重为正值或空值的图：Dijkstra 算法

- **定义**

给定一个有向图，其所有弧的权重都是正值或空值，我们在一个源点和一个目标节点之间寻找最短路径。

- **复杂度**

一个暴力实现的复杂度是 $O(|V|^2)$，用一个优先级队列优化后，可以让复杂度降低到 $O(|E|\log|V|)$。通过斐波那契优先级队列，我们能获得更低的复杂度 $O(|E|+|V|\log|V|)$，但为了实现优化而付出的努力过大，有点得不偿失。

- **算法**

我们仍采用本书 8.1 节的形式。Dijkstra 算法维护了一个顶点集合 S，我们已经计算好了从源点到这些顶点的最短路径，所以 S 一定是黑色顶点的集合。刚开始，S 只包含源点本身。另外，算法维护一个以源点为根的最短路径树来覆盖 S。我们用 prec[v] 来记录 v 的前驱顶点，用 dist[v] 来记录计算所得的距离（图 8.3）。

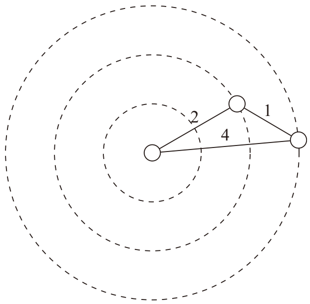


图 8.3　Dijkstra 算法用一个以顶点为圆心逐渐扩大的同心圆来捕捉顶点。由于没有距离为 1 的顶点，因而算法通过优先级队列，直接跳到最近的距离为 2 的顶点

然后来看 S 边缘的边。更准确地说，我们考虑弧 (u, v)，其中 u 在 S 内而 v 不在 S 内。这些弧定义了由一条从源点出发经过 u，并通过弧 (u, v) 到达 v 的最短路径。这条路径的权重即弧 (u, v) 的优先级。如 8.1 节的解释，算法从优先级队列中抽取一个优先级最小的弧 (u, v) 来定义最短路径 s-v，然后把弧 (u, v) 添加到最短路径树中，把 v 添加入 S 中，并继续迭代。

- **优先级队列**

算法的核心内容是优先级队列。这是一个能添加元素并抽取最小元素的元素集合而成的数据结构。这种结构通常使用堆来实现（即最小堆，见 1.5.4 节），此处的运算成本是与集合大小相关的对数。

- **小优化**

为了更高效，不要在优先级队列中保存那些不能引向最短路径的弧。为此，我们在每次向队列中添加通向 v 的弧时，应当更新 dist[v]。因此在检查一条弧时，我们就能确定是否有一个弧优于现有通向 v 的最优路径。

- **实现细节**

以下代码能计算出通向所有目标的最短路径，而且省略了在调用函数时指定某一特定目标参数的步骤。

在队列中，我们为每条离开弧 (u, v) 保存数据对 (d, v)，其中 d 是与弧相关的路径长度。

一个顶点可能在队列中以不同权重出现多次。但是，一旦该顶点第一次被抽取出来，就会被标注为黑色，而该顶点的其他相关记录会在抽取时被忽略。

```
from heapq import heappop, heappush

def dijkstra(graph, weight, source=0, target=None):
    n = len(graph)
    assert all(weight[u][v] >= 0 for u in range(n) for v in graph[u])
    prec = [None] * n
    black = [False] * n
```

```
    dist = [float('inf')] * n
    dist[source] = 0
    heap = [(0, source)]
    while heap:
        dist_node, node = heappop(heap)              # 最近的顶点
        if not black[node]:
            black[node] = True
            if node == target:
                break
            for neighbor in graph[node]:
                dist_neighbor = dist_node + weight[node][neighbor]
                if dist_neighbor < dist[neighbor]:
                    dist[neighbor] = dist_neighbor
                    prec[neighbor] = node
                    heappush(heap, (dist_neighbor, neighbor))
    return dist, prec
```

- **变种**

如果我们有一个能改变元素优先级的优先级队列，正如 1.5.4 节中介绍的那样，那么 Dijkstra 算法的实现就可以略微简化。我们不再把弧存入队列，而只保存顶点——实际上是图中所有顶点，并且一个顶点 v 的优先级就是 dist[v]。其实，队列保存了格式为 (dist[v], v) 的数据对，并用字典序比较它们。当发现一条通向 v 的更好路径时，我们把与 v 相关的数据对替换为一个更短的路径长度。变种的好处在于，我们不必再将顶点标注为黑色。对于每个顶点 v，队列仅包含一个与 v 相关的数据对。当 (dist[v], v) 从队列中被抽取时，我们就知道找到了通向 v 的最短路径。

```
from tryalgo.our_heap import OurHeap

def dijkstra_update_heap(graph, weight, source=0, target=None):
    n = len(graph)
    assert all(weight[u][v] >= 0 for u in range(n) for v in graph[u])
    prec = [None] * n
    dist = [float('inf')] * n
    dist[source] = 0
    heap = OurHeap([(dist[node], node) for node in range(n)])
    while heap:
        dist_node, node = heap.pop()                 # 最近的顶点
        if node == target:
            break
        for neighbor in graph[node]:
            old = dist[neighbor]
            new = dist_node + weight[node][neighbor]
            if new < old:
                dist[neighbor] = new
                prec[neighbor] = node
                heap.update((old, neighbor), (new, neighbor))
    return dist, prec
```

8.4 随机权重的图：Bellman-Ford 算法

- **定义**

这个问题允许图中弧上的权重为负值。假如存在一个从源点出发并能通过目标顶点的负权重环，那么源点到目标顶点的距离为 $-\infty$。当任意多次通过这个环时，我们反而会得到一条从源点出发的距离极小的路径[①]。下面介绍的算法能发现这一异常状况。

复杂度： 使用动态规划情况下是 $O(|V|\cdot|E|)$。

- **算法**

核心操作是释放距离（图 8.4），即对于每条弧 (u, v)，测试用该弧能否减少从源点到顶点 v 的距离。d_u+w_{uv} 是距离 d_v 的一个候选值。这一操作通过两个嵌套循环来完成，内层循环释放了通过每条弧到达顶点的距离；外层循环该操作，并执行一定次数。我们可以证明，在 k 次外层迭代后，能为每个顶点 v 计算出从源点到 v 且最多经过 k 条弧的最短路径。这个结论在 $k=0$ 时是正确的。对 $k=1,\cdots,|V|-1$ 的情况，用 $d_k[v]$ 来记录该距离时，我们有：

$$d_{k+1}[v]=\min_{u:(u,v)\in E} d_k[u]+w_{uv}.$$

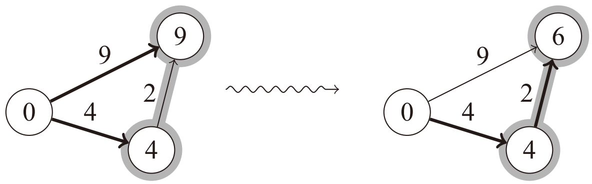


图 8.4 一个使用灰色边释放距离的例子。用这条边让源点到目标顶点的路径更短[②]

- **负环检测**

首先考虑图不包含负环的情况。在此情况下，所有最短路径都很简单，只需 $|V|-1$ 次循环迭代，就能确定所有通向目标的路径距离。因此，如果在 $|V|$ 次迭代中发现了一个变化，这表明存在着一个负环，并且是能从源点到达目标的负环。实现会返回一个布尔值来指出是否存在这样一个环。

---

① 因为环的权重是负值，所以多绕几圈反而让路径变短。——译者注

② 从源点 0 开始，经过一条权重为 9 的弧，到达右上角目标顶点；后变成通过右下角灰色顶点，总距离变成了 6。每个顶点上标注的数字是它到源点的距离。——译者注

```
def bellman_ford(graph, weight, source=0):
    n = len(graph)
    dist = [float('inf')] * n
    prec = [None] * n
    dist[source] = 0
    for nb_iterations in range(n+2):
        changed = False
        for node in range(n):
            for neighbor in graph[node]:
                alt = dist[node] + weight[node][neighbor]
                if alt < dist[neighbor]:
                    dist[neighbor] = alt
                    prec[neighbor] = node
                    changed = True
        if not changed:                     # 固定点
                return dist, prec, False
    return dist, prec, True
```

8.5　所有源点 – 目标顶点对：Floyd-Warshall 算法

- **定义**

给定一个在弧上有权重的图，我们希望计算每个顶点对之间的最短路径（图 8.5）。同样，问题只在图中不存在负权重环的情况下成立，而算法可以检测到这一异常情况。

$$a_{uv} = \sum_{k=0}^{n-1} b_{uk} \times c_{kv} \qquad W_{uv}^{(\ell+1)} = \min_{k=0}^{n-1} W_{uk}^{(\ell)} + W_{kv}^{(\ell)}$$

$$A = BC \qquad W^{(\ell+1)} = W^{\ell} W^{\ell}$$

图 8.5　Floyd-Warshall 算法可被视为热带代数[①] 下的矩阵乘法 ($\mathbb{R}$, min, +)。但我们不能使用它的快速乘积算法，因为这里只有一个半环

复杂度：使用 Floyd-Warshall 算法，复杂度为 $O(n^3)$

- **算法**

顶点间的距离是以动态规划来计算的（图 8.6）。对于每个 $k = 0, 1, \cdots, n$，我们计算一个矩形矩阵 W_k，并用 $W_k[u][v]$ 保存从 u 到 v 且仅经过下标严格小于 k 的中间顶点的最短路径长度，这些中间顶点编号为从 0 到 $n-1$。因此，对于 $k = 0$，矩阵 W_0 仅包含弧的权重；对于不存在进入弧 (u, v) 的情况，保存 $+\infty$。矩阵的更新基于一个简单原则：一条从 u 到 v 并经过顶点 k 的最短路径由一条从 u

① 热带数学（tropical mathematics）由巴西数学家、计算机科学家 Imre Simon 于 1980 年代提出并发展，是一种分片线性化的代数几何。——译者注

到 k 的最短路径和一条从 k 到 v 的最短路径组成。因此对于 $k,u,v \in \{0,\cdots,n-1\}$ ，我们计算：

$$W_k = 1[u][v] = \min\{W_k[u][v], W_k[u][k] + W_k[k][v]\}$$

对于相同的 k，它作为下标和在矩阵中代表的含义是一致的：为了计算 $W_{k+1}[u][v]$，可以考虑通过 k 的路径。

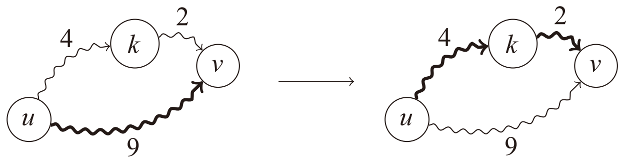


图 8.6　通过顶点 *k* 释放距离，能缩短从 *u* 到 *v* 的路径长度

- **实现细节**

以下算法维护了一个唯一数组 W 作为连续的 W_k，并使用参数来修改这个权重矩阵。算法实现能检测出是否存在负环，并在出现负环的情况下返回 `False`。

```
def floyd_warshall(weight):
    V = range(len(weight))
    for k in V:
        for u in V:
            for v in V:
                weight[u][v] = min(weight[u][v],
                                   weight[u][k] + weight[k][v])
    for v in V:
        if weight[v][v] < 0:       # 检测到了负环
            return True
    return False
```

- **检测负环**

当且仅当 $W_n[v][v] < 0$ 时，存在一个通过顶点 v 的负环。然而，正如 Hougardy 在参考文献 [7] 中介绍的，我们更推荐 Bellman-Ford 算法来检测负环。因为如果存在负环，当顶点数量较大时，用 Floyd-Warshall 算法计算出来的绝对值可能会呈指数阶增长，直到造成变量的存储空间溢出。

8.6　网格

- **问题**

给定一个矩形网格，其中有些格子是可以通过的，我们希望找到从入口到出口的最短路径。

● **算法**

这个问题可以采用一个简单方法——在网格图上的广度优先遍历算法。但是，相对于显式地建立一个图，在网格上直接做遍历反而更加容易。给定网格的描述方式是一个二维数组，其中 # 字符用来表示不可通过的格子，空字符表示可通过的格子。算法的实现用二维数组来标注已访问过的顶点，避免新造一个额外的数据结构。那么，被访问过的格子将包含字符→、←、↓、↑，注明了从源点出发需要通过路径的前驱顶点。

```
def dist_grid(grid, source, target=None):
    rows = len(grid)
    cols = len(grid[0])
    dir = [(0, +1, '>'), (0, -1, '<'), (+1, 0, 'v'), (-1, 0, '^')]
    i, j = source
    grid[i][j] = 's'
    Q = deque()
    Q.append(source)
    while Q:
        i1, j1 = Q.popleft()
        for di, dj, symbol in dir:           # 探索所有方向
            i2 = i1 + di
            j2 = j1 + dj
            if not(0 <= i2 and i2 < rows and 0 <= j2 and j2 < cols):
                break                        # 越过了网格的边界
            if grid[i2][j2] != ' ':          # 不可通过或已访问过的格子
                continue
            grid[i2][j2] = symbol            # 标注已经访问
            if(i2, j2) == target:
                grid[i2][j2] = 't'           # 到达目标
                return
            Q.append((i2, j2))
```

● **变种**

对于共享一个角落的格子来说，上述实现能很容易被修改，从而实现斜线移动。把一个六边形网格变为一个有特殊相邻关系的正方形网格，也能用相同方法来处理。图 8.7 演示了上述变换。

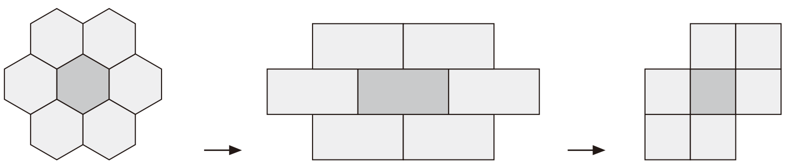


图 8.7　使用正方形网格来表示六边形网格

8.7 变种问题

最短路径是一个重要问题，以下将介绍几种经典变种。

8.7.1 无权重图

在一个无权重图中，只需要执行一次广度优先遍历就可以确定最短路径。

8.7.2 有向无环图

一次拓扑排序就能以可接受顺序处理所有顶点（见 6.8 节）：为了计算从源点到顶点 v 的距离，可以首先计算到所有 v 的前驱顶点的距离，然后再使用一个简单的动态规划算法来得到答案。

- **应用**

在从家走到办公室的路上，我想先走上坡路再走下坡路，以便先运动、后休息。为此，我找到一张城市的建模图，其中的顶点是有高度值的区域交叉点，而边是有长度值的道路。

8.7.3 最长路径

上述动态规划算法可以应用在有向无环图上。对于一个通用图，最长路径问题旨在找到一条从源点到目标顶点，而且只通过每个顶点一次的最长路径。这是个 NP 复杂问题，已知任何算法都不能在多项式时间内解决该问题。如果顶点数量很少，比如在 20 个左右，那我们可以在顶点集合 S 的子集中使用动态规划算法，并计算 $D[S][v]$，从源点到 v 的最长路径一定只使用集合 S 中的顶点作为中间节点。如此一来，对于所有非空集合 S，有如下关系：

$$D[S][v] = \max_{u \in S} D[S \setminus u][u] + w[u][v]$$

其中 S\u 是集合 S 去掉顶点 u 后的集合，$w[u][v]$ 是弧 (u, v) 的权重。因此，基本情况如下：

$$D[\varnothing][v] = \begin{cases} w[u][v], \text{如果存在弧}(u,v) \\ -\infty, \text{否则} \end{cases}$$

8.7.4 树中的最长路径

通常，看似复杂的问题在树中会变得简单，因为子树可以利用动态规划算法找到解决方案。这也是树中最长路径问题的情况，10.3 节介绍了一个线性复杂度算法。

8.7.5　最小化弧上权重的路径

当我们不希望路径上的边的总权重**最小**，而希望它**最大**时，使用并查集数据结构会更简单。从一个空的图开始，按权重升序向图中添加边，直到源点和目标顶点位于同一个连通分量中。有向图也存在一个类似解决方案，但实现过程相当复杂。

8.7.6　顶点有权重的图

考虑一个弧上没有权重而顶点有权重的图，目的是找到一条从源点到目标顶点的路径，让该路径通过的所有顶点的权重总和最小。我们把每个顶点 v 替换为两个顶点 v^- 和 v^+，这两个顶点被权重为 v 的弧连接，同时把每个弧 (u, v) 替换为弧 (u^+, u^-)，这是本问题就等价于上一个问题。

8.7.7　令顶点上最大权重最小的路径

当一条路径的权重被定义为“路径上中间节点的权重最大值”时，可以使用 Floyd-Warshal 算法。只需把所有顶点按照权重升序排列，并在找到从源点到目标顶点的路径时，立即结束迭代。

维护一个保存着连通性的布尔型数组 $C_k[u][v]$，数组表示仅使用下标小于 k 的中间节点时，是否存在一条从 u 到 v 的路径。更新方法如下：

$$C_k[u][v] = C_{k-1}[u][v] \lor (C_{k-1}[u][k] \land C_{k-1}[k][v])$$

8.7.8　所有边都属于一条最短路径

给定一个有权重的图、一个源点 s 和一个目标顶点 t，s 和 t 之间可能存在多条最短路径。目标是确定所有边是否属于一条最短路径。为此，我们使用两次 Dijkstra 算法来处理 s 和 t，计算从源点 s 出发到顶点 v 的距离 $d[s, v]$，以及从顶点 v 到目标顶点 t 的距离 $d[v, t]$。然后，当且仅当

$$d[s, u] + w[u, v] + d[v, t] = d[s, t]$$

存在一条边 (u, v) 属于一条最短路径，其中 $w[u, v]$ 是边的权重。

- **无向图的变种问题**

Dijkstra 算法只能处理一个给定源点，而不是一个给定的目标顶点。因此，在计算所有 v 到 t 的距离 $d[v, t]$ 时，需要暂时把弧反转。

第 9 章　耦合性和流

一般情况下，组合优化的核心部分由耦合性和流问题组成。这两个问题彼此相连，存在多个变种问题。算法的原理是对一个解反复优化：从起初的空解，最终得到一个最优解。

假设在图 9.1 的二分图[①]中，我们希望确定一个**完美匹配**，也就是说，把图中左侧所有顶点与唯一一个右侧顶点相关联。图中的边表示哪种关联是可以实现的。如果我们从关联 u_0 和 v_0 开始，就会被阻挡。为实现一个完美匹配，必须解开这个关联。这一原理将在后面章节中解释。

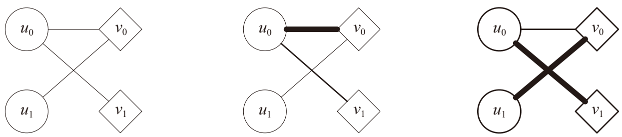


图 9.1　逐渐建立起一个完美匹配

这里介绍的流算法需要满足以下条件：对于每一条弧 (u, v)，存在其逆向弧 (v, u)。算法会首先调用一个方法，以便在必要时用权重为 0 的逆向弧把图补全，借此测试对于每条弧 (u, v) 是否有 u 存在于 v 的相邻节点列表。其时间复杂度为 $O(|E| \cdot |V|)$，在当前情况下不可忽略。

```python
def add_reverse_arcs(graph, capac):
    for u in range(len(graph)):
        for v in graph[u]:
            if u not in graph[v]:
                graph[v].append(u)
                capac[v][u] = 0
```

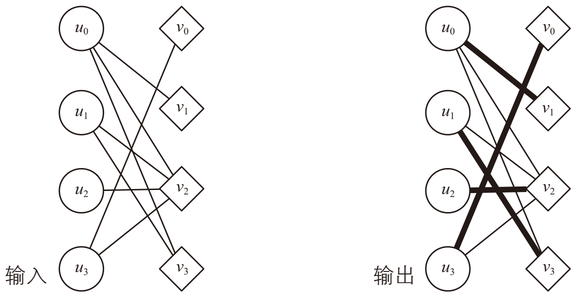
- **复杂度**

在下表中，我们假设对二分图 G(U, V, E) 有 $|U| \leqslant |V|$，用 C 来记录最大容量。

① 二分图又称双分图、二部图、偶图，指顶点可以分成两个不相交的集 U 和 V（U 和 V 皆为独立集），使得在同一个集内的顶点不相邻（没有共同边）的图。——译者注

耦合性		
无权重二分图	$O(\|V\|\cdot\|E\|)$	增量路径算法
有权重二分图	$O(\|V\|^3)$	Kuhn-Munkres 算法
有偏好的二分图	$O(\|V\|^2)$	Gale-Shapley 算法
流		
有界容量	$O(\|V\|\cdot\|E\|\cdot\|C\|)$	Ford-Fulkerson 算法
有界容量	$O(\|V\|\cdot\|E\|\cdot\log C)$	二进制阻塞流算法
随机容量	$O(\|V\|\cdot\|E\|^2)$	Edmonds-Karp 算法
随机容量	$O(\|V\|^2\cdot\|E\|)$	Dinic 算法
割		
随机图	$O(\|V\|^2\cdot\|E\|)$	Dinic 算法
随机图	$O(\|E\|\log\|V\|)$	Dijkstra 算法

9.1　二分图最大匹配



- **应用**

在 n 个房间之间修建 n 条走廊，给 n 个工人分配 n 项任务……二分图最大匹配问题的应用范围非常广。9.2 节还将介绍一个有权重的问题。由于这些问题通常使用二分图来建模，因而我们仅介绍此类情况——当然，这些问题也可以用其他图类数据结构来解决。

- **定义**

设二分图 G(U, V, E)，且 $E \subseteq u \times v$。匹配是集合 $M \subseteq E$，且在 M 中不存在拥有公共顶点的两条边。目的是找到一个最大基数的匹配。对于给定集合 M，当一个顶点位于 M 中的一条边上时，我们称该顶点**被匹配**，否则称之为**自由顶点**①。

① 此处设定的二分图 G(U, V, E) 中，U 和 V 分别是二分图两个不相交的独立集，而 E 是连接这两个独立集的边的集合；U×V 是把所有 U 中的顶点和所有 V 中的顶点一一相连的边的集合，因此 E 一定是 U×V 的子集，或者二者相等。

- **关键测试**

一个最容易想到的优化解方法就是从穷举法开始，直到找到最优解。为了优化一个匹配，需要观察两个匹配间的**对称差**[①]。对于一个实线的匹配 M 和一个虚线的匹配 M′，二者的对称差 M⊕M′ 由 M\M′ ∪ M′\M 来定义，后者由实、虚交替的边构成的路径和环组成。通过参数计数可以发现，当 $|M'| > |M|$ 时，总存在一条虚线开头和虚线结尾的实虚交替路径 P（图 9.2）。

路径 P 在一个自由顶点开始和结束，并在属于 M 和不属于 M 的边之间交替。我们把这种路径称作**增广路**，因为 M⊕P 的差异是 M 增加一条边后的一个匹配。

因此，如果 M 还不是最大基数匹配，那么对于 M 存在一条增广路。更准确地说，如果存在一个匹配 M′ 使得 $|M'| > |M|$，而且一个顶点 $u \in$ U 在 M′ 中是配对顶点，但在 M 中是自由顶点，那么对于 M 一定存在一条从 u 出发的增广路 P。

- **复杂度为 $O(|U|\cdot|E|)$ 的算法**

上述结果引出了第一个算法。从一个空匹配 M 开始，寻找 M 的一条增广路径 P，并用 M ⊕ P 来替换 M，直到找不到 P 为止。通过深度优先遍历很容易就能找到这样一条增广路径。只需从 U 的一个自由顶点开始，考虑它还没有被访问过的所有相邻节点。如果 v 尚未被匹配，那么从根节点到 v 的路径是一条增广路径；如果 v 已经与一个顶点 u' 匹配，那么从 u' 继续遍历。找到并应用一条增广路的算法复杂度是 $O(|E|)$，现在必须在最多 $|U|$ 个顶点上重复这一操作，于是本算法的时间复杂度应当是 $O(|U|\cdot|E|)$。

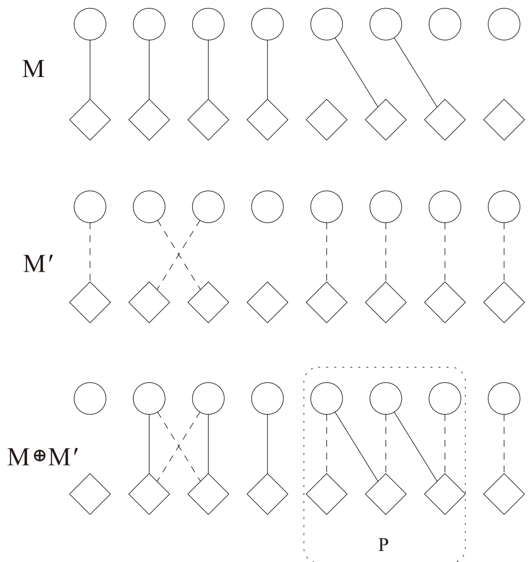


图 9.2　对称差 M⊕M′ 由一个颜色交替环、一条 M′ 的增广路 P 和两条 M 的增广路组成

① 对称差为两个集合的并集减去交集。——译者注

● **实现细节**

在下面介绍的实现[①]中，我们用一个数组 m 来编码一个匹配，它为每个 $v \in V$ 顶点关联了一个 U 中与之匹配的顶点；否则，假如 v 是自由顶点，$m[v]$ = None。

图由数组 E 给出，其中 $E[u]$ 是 u 相邻顶点的列表。给定顶点集合 U 和 V 的格式应该是 [0,1,⋯, $|U|-1$] 和 [0, 1, ⋯, $|V|-1$]。

```
def augment(u, bigraph, visit, match):
    for v in bigraph[u]:
        if not visit[v]:
            visit[v] = True
            if match[v] is None or augment(match[v], bigraph, visit, match):
                match[v] = u                        # 找到了增广路
                return True
    return False

def max_bipartite_matching(bigraph):
    n = len(bigraph)   # U 和 V 的范围相同
    match = [None] * n
    for u in range(n):
        augment(u, bigraph, [False] * n, match)
    return match
```

● **其他算法**

此外，Hopcroft Karp 算法可用来在时间 $O(\sqrt{|V|} \cdot E)$ 内解决二分图的最大匹配问题。其原理是在同一次遍历中找到多条增广路，并从中选择最短的增广路。Alt、Blum、Mehlhorn 和 Paul（见参考文献 [2]）也找到了一个有趣的算法处理稠密图（有很多边），其时间复杂度为 $O(|V|^{1.5}\sqrt{|E|/\log|V|})$。但是，所有这些算法实现起来都相当复杂，没有本节中介绍的算法如此快速、实用。

● **二分图中的最小覆盖问题**

给定一个二分图 G(U, V, E)，我们寻找一个最小的基数集合 $S \subseteq U \cup V$，使得每条边 $(u, v) \in E$ 至少拥有一个 S 中的末端顶点，因此 $u \in S$ 或 $v \in S$。由于一个匹配的每条边必须至少拥有一个 S 中的末端顶点，匹配的最大值就应该是最小覆盖的下限值。Konig 定理证明了，实际上二者最优值相等。

定理的证明极具建设性，给出了一个算法，基于图 G(U, V, E) 的一个最大匹配来找到一个最小覆盖。设没有被 M 匹配的 U 中顶点集合 Z，在 Z 中添加通过交替路径能到达的所有顶点，我们定义以下集合：

$$S = (U \backslash Z) \cup (V \cap Z)$$

集合 Z 的结构说明，对于所有边 $(u, v) \in M$，如果 $v \in Z$，那么有 $u \in Z$。同样对于 $u \in Z$，由于 u 起初不与自由顶点 U 同在 Z 中，u 随着匹配边才被添加入 Z 中，因此 $v \in Z$。

这证明了对于每条属于匹配 M 的边 (u, v)，其末端顶点要么全都在 Z 中，要么都不在 Z 中。所

① 为什么只有 V 中的顶点在被访问时被标记？大家可以想一想。

以，所有匹配的边都有且仅有一个末端顶点在 S 中，且 $|S| \geqslant |M|$。

我们可以证明 S 覆盖了图的所有边。设 (u, v) 是图的一条边。如果 $u \notin Z$，边被覆盖，那么设 $u \in Z$。如果 (u, v) 不是匹配的一条边，那么 Z 的最大字符数量使得 v 必须在 Z 中，因此 v 也就在 S 中。如果 (u, v) 在 M 中，那么通过前面的论证有 $v \in Z$，因此 $v \in S$。这就证明了 S 是一个顶点的覆盖，这意味着 $|S| \leqslant |M|$，故而证明了结论 $|S| = |M|$。

9.2 最大权重的完美匹配：Kuhn-Munkres 算法

	v_0	v_1	v_2	v_3
u_0	3	3	1	2
u_1	0	4	–8	3
u_2	0	2	3	0
u_3	–1	1	2	–5

输入

	v_0	v_1	v_2	v_3
u_0	**3**	3	1	2
u_1	0	4	–8	**3**
u_2	0	**2**	3	0
u_3	–1	1	**2**	–5

输出

- **应用**

在一幢教学楼里，有 n 节课程要分配给 n 位教师。每位教师用一个权重表显示自己对课程安排的偏好。这个权重被标准化成总和为 1 的一系列数字，好让每位教师的偏好值一致。目的是找到课程和教师之间的二分图，使得所有课程分配的权重和最大。这就是在一个最大收益二分图中找完美匹配的问题。

- **定义**

对于一个二分图 G(U, V, E)，在每条边上有权重 w: E → R。在不丧失普适性的情况下，假设 |U| = |V|，而且图是个完全图，即 E = U ◊ V。目的是找到一个完美匹配 $M \subseteq E$，使得权重总和（又称收益）$\sum_{e \in M} w(e)$ 最大化[①]。

- **变种**

这个问题的一个变种是计算最小成本的完美匹配。在这种情况下，只需改变权重的正负号，并采用最大化收益的思路即可。如果 $|U| > |V|$，只需在 V 中加入新的顶点，并使用权重为 0 的边连接 U 中所有顶点。由此，新图的完美匹配和原图的最大匹配就有了相关性。如果新图不是完全图，只需使用权重为 $-\infty$ 的边来补全即可，这些边在寻找最优解的过程中一定不会被选中。

复杂度： Kuhn-Munkres 算法（又称匈牙利算法）的时间复杂度为 $O(|V|^3)$。

---

① 在完全图中，所有顶点都有且仅有一条边互相连接。在判断收益或成本的时候，可以把原本不相连的边标注为连接，但权重为无穷小或无穷大，因此，把一个普通二分图拓展成为完全图并不会丢失普适性。——译者注

● **算法**

Kuhn-Munkres 算法属于原始 – 对偶类算法，使用了线性规划的建模方法。为方便理解，以下介绍不会使用线性规划的术语。

主要思路是考虑一个关联问题，即**最小有效顶标**问题。顶标是顶点上的权重 l，当所有边 (u, v) 都满足

$$l(u) + l(v) \geqslant w(u, v) \tag{9.1}$$

它们是有效的顶标。关联问题旨在找到总权重最小的有效顶标。对一个匹配所有边 (u, v) 上的不等式求和，很容易看到，有效顶标的权重之和大于完美匹配的权重。

顶标 l 定义的集合 E_l 由满足以下算式的所有边 (u, v) 组成

$$l(u) + l(v) = w(u, v)$$

仅包含这些边的图被称作等价图。如果我们考虑有效顶标 l 和有效顶标集合上的一个完美匹配 $M \subseteq E_l$，那么 $\sum_{E_l} l$ 的值等于 $|M|$。由于 $\sum_{E_l} l$ 的值是所有完美匹配的最大值，这证明了 M 的权重最大。

现在要建立一个数值对 (l, M)。在所有情况下，算法都有有效顶标集合 l 和一个匹配 $M \subseteq E_l$。算法用循环的方式扩展匹配 M；如果不能扩展了，算法会优化顶标，即减小顶标值的和[①]。

● **扩展匹配**

为了扩展匹配，必须在等价图中找到一条增广路，所以需要建立一棵交替树。我们从选择一个自由顶点 u_i 开始，$u_i \in$U 且没有匹配。顶点 u_i 作为树的根节点。树在 U 和 V 的顶点之间交替，也在 $E_l \backslash$M 和 M 的边的不同层级之间交替。通过深度遍历算法就能建立这样一棵交替树。一旦 v 的一个叶子节点成为自由顶点，那么从 u_i 到 v 的路径就是一条增广路，而且通过这条路径有可能实现对 M 的扩展匹配，并把 $|M|$ 增加 1。在这种情况下，建立交替树的过程就终止了。

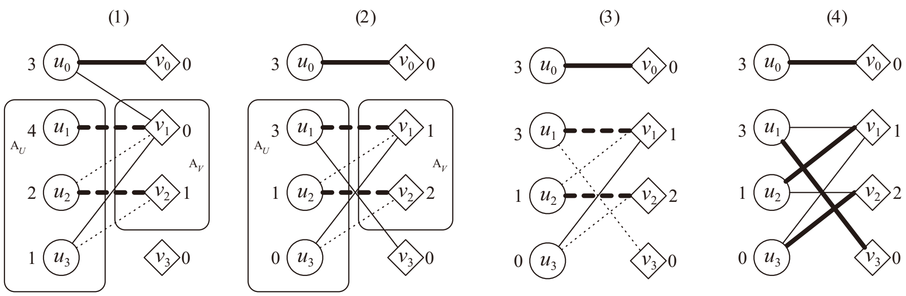


图 9.3　Kuhn-Munkres 算法的过程。(1) 本章一开始给出了用权重矩阵描述的二分图，图 (1) 中标注了每个顶点的权重，记录了等价图的每条边，并用粗线表示了一个匹配的所有边，用虚线表示了根节点为 u_3 的最大交替树的边。这棵树覆盖了顶点集合 A_U 和 A_V，且无法被扩展。(2) 优化顶标，添加了一条新的边 (u_1, v_3)。(3) 交替树包含了的自由顶点 v_3。(4) 匹配沿着从 u_3 到 v_3 的路径扩展

① 优化顶标同样也是降低总成本，这就是本算法的最终目的。——译者注

- **交替树的性质**

现在考虑匹配没能被扩展的情况。我们已经建立了一个交替树 A，其根节点是自由顶点 u_i；同时，集合 V ∩ A 中任意一个顶点都不是自由的。我们把 U 中被交替树覆盖的顶点集合称作 A_U，V 中被 A 覆盖的顶点集合称作 A_V。于是有 $|A_U| = 1+|A_V|$。换一种形式来说，A_U 由树的根节点以及与 A_V 每个顶点的匹配顶点组成。

- **优化顶标**

优化顶标时必须要谨慎：首先顶标必须是有效的；其次树 A 必须留在等价图中。因此，一旦我们把根节点为 u_i 的顶标减少一个值 $\Delta > 0$，就必须要把树中的后代节点增加一个相同的值，这样一来，我们就能在处理一个节点时把其后代节点减少 Δ。最终，优化就是把 A_U 中所有顶点的顶标值减少 Δ，并把 A_V 中所有顶点的顶标值增加 Δ。标注的总值一定是严格减少的，因为 $|A_U| > |A_V|$。我们发现不等式 (9.1) 左右两边的差——称作**裕度**——在 $(u,v)\in A$ 或 $uv \notin A$ 的时候被保留，所以树 A 能够留在等价树中。为了保证标签有效，只需关注边 (u, v)，其中 $u\in A_U$ 和 $v \notin A_V$。因此，我们可以确定 Δ 是这些边上的最小裕度（图 9.3）。

- **进步**

当在一条从 A_U 到 $V\backslash A_V$ 的额外边进入等价图时，以下方法会生效，尤其在这条边确定了最小裕度 Δ 时，因为其裕度变成了 0。注意，图中其他边可能会消失，但这对本算法来说不重要。

- **初始化**

为了让算法运行，我们从有效顶标 l 和空匹配 M 开始。为了简化阐述，对于所有 $v\in V$，我们选择 $l(v)=0$，而对于所有 $u\in U$，选择 $l(u) = \max_{v\in V} w(u, v)$。

- **算法实现时间复杂度为 $O(|V|^4)$**

算法的实现是一个对 $u_i\in U$ 顶点的外部循环，其中的常量表示所有已遇到的顶点都被 M 匹配。这个循环的复杂度是 $O(|V|)$。接下来，对每个自由顶点 u_i 建立一棵交替树，以此尝试建立匹配，或在必要情况下优化顶标。建立交替树的过程的复杂度是 $O(|V|^2)$。

每次优化顶标以后，交替树会增长，特别是 $|A_V|$ 会严格增长，但 $|A_V|$ 的上限是 $|U|$，因此匹配的增长成本是 $O(|V|^2)$，而完整的时间复杂度是 $O(|V|^4)$。[①]

```
def improve_matching(G, u, mu, mv, au, av, lu, lv):
    assert not au[u]
    au[u] = True
    for v in range(len(G)):
        if not av[v] and G[u][v] == lu[u] + lv[v]:
            av[v] = True
            if mv[v] is None or \
```

① Kuhn-Munkres 算法理解起来有一定难度，建议读者尝试结合程序代码来理解整个证明算法时间复杂度的逻辑过程，以及实现算法所使用的数据结构。——译者注

```
                improve_matching(G, mv[v], mu, mv, au, av, lu, lv):
                 mv[v] = u
                 mu[u] = v
                 return True
    return False

def improve_labels(G, au, av, lu, lv):
    U = V = range(len(G))
    delta = min(min(lu[u] + lv[v] - G[u][v]
                for v in V if not av[v]) for u in U if au[u])
    for u in U:
        if(au[u]):
           lu[u] -= delta
    for v in V:
        if(av[v]):
           lv[v] += delta

def kuhn_munkres(G):                    # 复杂度为 O(n^4) 的最大收益的完美匹配
    assert len(G) == len(G[0])
    n = len(G)
    mu = [None] * n                     # 空匹配
    mv = [None] * n
    lu = [max(row) for row in G]        # 平凡标签
    lv = [0] * n
    for u0 in range(n):
        if mu[u0] is None:              # 自由顶点
            while True:
                au = [False] * n        # 空的交替树
                av = [False] * n
                if improve_matching(G, u0, mu, mv, au, av, lu, lv):
                    break
                improve_labels(G, au, av, lu, lv)
    return(mu, sum(lu) + sum(lv))
```

- **实现细节**

我们用从 0 到 n-1 的整数为 U 和 V 中的顶点进行编码。为此，必须把 l 编码到数组 lu 和 lv 中，来对应集合 U 和 V。同样，mu 和 mv 保存着匹配结果，并在当 $u\in$U、$v\in$V 且二者匹配的时候，有 $mu[u] = v$, $mv[u] = v$。

自由顶点以 $mu[u]$ = None 或 $mv[v]$ = None 来标记。最终，布尔型数组 au 和 av 明确了一个顶点是否被交替树覆盖。

- **算法实现的时间复杂度为 $O(|V|^3)$**

为了获得立方级的时间复杂度，必须维护一个 margeVal 数组来简化顶标优化过程中对裕度 Δ 的计算。那么，对于每个满足 $v\in V\backslash A_V$ 的顶点有：

$$\text{margeVal}_v = \min_{u\in A_u} l(u) + l(v) - w(u,v)$$

让以上算式最小化的顶点 u 存储在另一个数组 margeArg 中。

这些数组让树的扩展过程更容易了，因为每次变动顶标后，不再需要从根节点开始构建整棵树。树的构建过程也不再采用深度优先遍历算法，而是以下述方式来选择一个新的边。

选择满足 $u \in A$、$v \notin A$ 且使裕度最小的边 (u, v)。当裕度非 0 时，我们可以按照前面的算法来更新顶标，结果是把 (u, v) 的裕度减小到 0。任何情况下，(u, v) 是一条可被添加到树中的边时，有两种情况要分别处理。对于还没有匹配的 v，我们已经找到了一条增广路；对于已和某个顶点 u' 配对的 v，我们可以把 (u', v) 添加到树 A 中，继而把 u' 添加入 A_U 中，同时将裕度在线性时间内更新，并重新开始。因此，找到一条增广路所需时间呈 4 次幂。一旦 U 中的一个顶点被匹配，它会被保留在 U 中，所以要寻找的只剩下 $|V|$ 条增广路径。实现最终的复杂度是 $O(|V|^3)$。

- **实现细节**

这一次，我们想知道 V 中的每个顶点是否是树的一部分？如果是，它们的前驱节点是谁？树不是通过深度优先遍历建立起来的，所以我们想知道的信息必须被存储在一个数组中。当扩展一个匹配时，我们借此能回溯到树的根节点。把节点 $v \in A_V$ 的前驱节点记为 $A_V[v]$，把树外节点 $v \in v \backslash A_V$ 记为 $A_V[v] =$ None。数组变量的首字母大写，以便与存储 A_U 的布尔型数组区分。

数组 marge 统一表示了数组 margeVal 和 margeArg，并保存数据对 (val, arg)。

```
def kuhn_munkres(G):                                # 最大收益完美匹配复杂度 O(n^3)
    assert len(G) == len(G[0])                      # 正方形矩阵
    n = len(G)
    U = V = range(n)
    mu = [None] * n                                 # 空匹配
    mv = [None] * n
    lu = [max(row) for row in G]                    # 平凡标签
    lv = [0] * n
    for root in U:                                  # 建立一个交替树
        n = len(G)
        au = [False] * n
        au[root] = True
        Av = [None] * n
        marge = [(lu[root] + lv[v] - G[root][v], root) for v in V]
        while True:
            ((delta, u), v) = min((marge[v], v) for v in V if Av[v] == None)
            assert au[u]
            if delta > 0:                           # 树已经完成
                for u0 in U:                        # 优化顶标
                    if au[u0]:
                        lu[u0] -= delta
                for v0 in V:
                    if Av[v0] is not None:
                        lv[v0] += delta
                    else:
                        (val, arg) = marge[v0]
                        marge[v0] = (val - delta, arg)
```

```
            assert lu[u] + lv[v] == G[u][v]
            Av[v] = u                          # 把 (u, v) 添加入集合 A
            if mv[v] is None:
                break                          # 找到了交替路
            u1 = mv[v]
            assert not au[u1]
            au[u1] = True                      # 把 (u1, v) 添加入集合 A
            for v1 in V:
                if Av[v1] is None:             # 更新裕度
                    alt = (lu[u1] + lv[v1] - G[u1][v1], u1)
                    if marge[v1] > alt:
                        marge[v1] = alt
        while v is not None:                   # 找到了交替路径
            u = Av[v]                          # 沿着路径向根节点
            prec = mu[u]
            mv[v] = u                          # 扩展匹配
            mu[u] = v
            v = prec
    return(mu, sum(lu) + sum(lv))
```

- **最大权重边的最小完美匹配**

定义

设一个二分图 (U, V, E)，边都标注了权重 w:E → Z，目的是找到一个完美匹配 M。但我们不是要最小化 M 中所有边的权重和，而是最小化 M 中一条边的最大权重，并在所有完美匹配上计算 $\min_M \max_{e\in M} w(e)$ 。

把问题简化为最大匹配问题

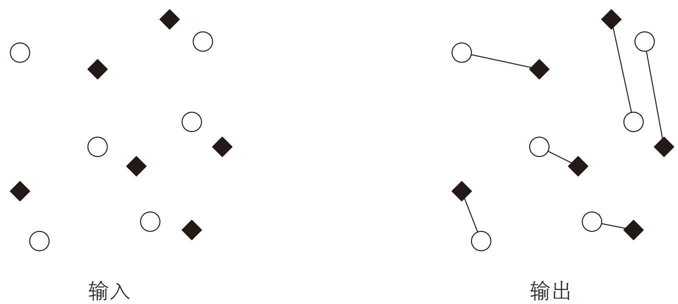
假设所有边都按照权重升序排序，对于一个给定的阈值 k，我们可以测试图中最前 k 条边中是否存在一个完美匹配。这一属性在 k 上是单调的，使用二分查找方式可以在时间区间 $[|V|, |E|]$ 内解决问题。

利用寻找完美匹配算法的特殊运行机制，我们还能把二分查找的时间复杂度再次节省 $O(\log|E|)$。

算法借助一条增广路来扩展当前匹配，以此构建一个完美匹配。这条增广路来自交替树森林[①]。我们把变量 k 初始化为 0，并维护一个匹配 M，以及一个由前 k 条边组成的交替树。当交替树构建完成，却仍没有找到增广路时，我们增大 k 值并把第 k 条边添加到图中，同时更新交替树。根据当前边的数量，每次扩展 M 都需要线性时间复杂度，于是，找到让图形成一个完美匹配的最小 k 值所需时间复杂度为 $O(|V| \cdot |E|)$。

① 整个图里面的所有交替树。——译者注

9.3 无交叉平面匹配



- **定义**

图中给定 n 个白色点和 n 个黑色点。假设输入不会退化，即不存在 3 个点共线。目的是在白色点和黑色点之间找到一个完美匹配，使得当把所有点与其右侧点连接时，所有连接线不相交。

- **关键测试**

假设在某个时刻，点都已形成了匹配，但仍存在连接线相交的情况。考虑连接线相交的两对匹配 u-v 和 u'-v'（图 9.4），当把它们的关联改成 u-v' 和 u'-v 时，连接线就不相交了。那么，这一操作能不能优化解呢？

我们注意到，在执行上述解除交叉的方法后，交叉线的数量反而可能会增加，如图 9.4 所示。因此，这不是优化解的正确方法。相反，所有连接线的总长度减少了。这个测试把我们带回第一种算法。

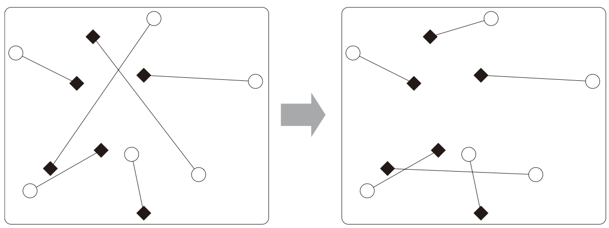


图 9.4 解除匹配中存在的交叉

- **不保证性能的算法**

在两部分中随机地匹配点。由于存在两条相交连线，因而需要用上述方法解除交叉。通过前面的论证，这个算法保证能找到一个解。根据我们的经验，在实践中该算法的性能很好。

- **减小完美匹配最小成本的算法复杂度为 $O(n^3)$**

定义一个完全二分图 G(U, V, E)，满足 U 中保存所有白色点，V 中保存所有黑色点，而且对于所有 $u \in$ A 和 $v \in$ V，边 (u, v) 的成本是两点间的欧氏距离。前面的论述证明，最小成本的完美匹配一

定没有交叉。因此，只需在图上应用 Kuhn-Munkres 算法就能解决问题。但是，我们意识到工作量太大，因为一个没有交叉的匹配不一定成本最小。

- **复杂度为 $O(n\log n)$ 的算法**

基于“火腿三明治”理论，有一个更好的算法。Banach 在 1938 年提出，当图中有 n 个黑点和 n 个白点，并且它们处于非退化位置（不存在 3 点共线）的时候，一定存在一条直线，使得直线两边有同样多的白点和黑点，精确地讲是直线每侧各 $n/2$ 个。事实上，如果 n 是奇数，这条直线一定会通过一个白点和一个黑点；如果 n 是偶数，直线不会通过任何点。

1994 年，Matousek、Lo 和 Steiger 提出的算法能在 $O(n)$ 时间内找到这条直线（见参考文献 [22]），但算法的实现比较复杂。

这会有什么帮助吗？在确保不存在 3 点共线的前提下，我们可以得到如下属性：n 为偶数时，分割线不会通过任何点；n 为奇数时，分割线一定会通过一个白点和一个黑点，此时就可以把二者匹配起来。无论如何，我们都可以在分割线切分的两个独立空间中用迭代法进行匹配。这个递归拆分过程的深度是 $O(\log_2 n)$，因此算法的最终复杂度是 $O(n\log n)$。

9.4　稳定的婚姻：Gale-Shapley 算法

- **定义**

假设有 n 位女性和 n 位男性，每位男性都对女性做了一个偏好排列，而女性也对男性做了同样的偏好排列。一次婚姻就是在男性和女性的二分图上形成一个完美匹配。假设不存在一个男性 i 和一个女性 j，令丈夫更偏好女性 j 而非自己的配偶，或者令妻子更偏好男性 i 而非自己的配偶，这次婚姻被称作**稳定**。目标是通过 $2n$ 个偏好列表找到一个稳定婚姻关系。解决方案不是唯一的。

复杂度：使用 Gale-Shapley 算法的复杂度为 $O(n^2)$。

- **算法**

算法从没有已婚夫妇的情况开始。然后，只要仍存在男性单身者，算法就会选择一个单身男性 i 和 i 最偏爱的女性 j。算法尝试让 i 和 j 结婚。如果 j 仍然单身，这个操作会被执行；如果 j 已经和一个男人 k 结婚，但她更偏好 i 而非 k。在这种情况下，k 只能回到单身男性的行列中[①]。

① 参照上述解除交叉法来解除一个匹配并连接另一个匹配。男性先与自己更偏好的女性匹配，但如果女性相对于这个匹配有更偏好的单身男性可选，那么就会选择自己更偏好的匹配；而失去匹配的男性就要重复进行这一操作，选择自己当下最偏好的女性继续尝试匹配。因此，女性每次都可以和自己更偏好的男性匹配，而男性每次重新确定的匹配都是比与原配更差的选择。——译者注

- **分析**

对于复杂度来说，所有配对 (i, j) 最多仅被算法考虑一次，那么确定每对夫妻的工作就是常量。为了确保有效，只需在算法过程中测试：(1) 一个给定女性与她更偏好的男性结婚；(2) 男性与他更不偏好的女性结婚。我们通过反证法来证明算法的有效性。假设最终存在一个男性 i 与一个女性 j′ 结婚，而一个女性 j 与一个男性 i′ 结婚，而且 i 相对于 j′ 更偏好 j，j 相对于 i′ 更偏好 i。通过测试 (2)，算法在某个时刻已经考虑了配对 (i, j)，但根据测试 (1)，算法应该没有将 i 与 j 匹配，也就是说，当算法考虑配对 (i, j) 时，j 本该已经与比起 i 更偏好的 k 结婚了。这和最终她（j）与自己相对于 i 更不喜欢的男人（i′）结了婚的事实矛盾。

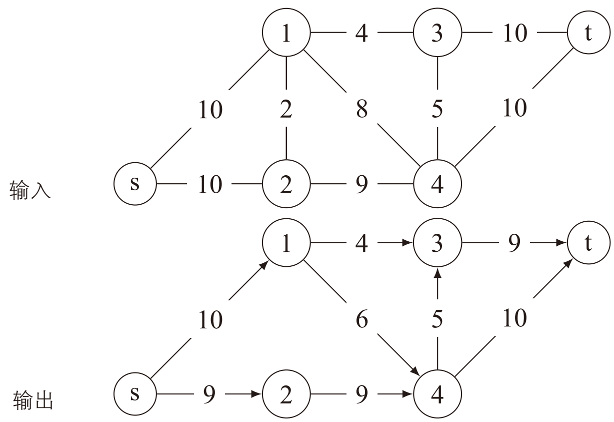
- **实现细节**

在实现中，男性和女性都被从 0 到 n-1 编号。输入数据由两个数组组成。数组 men 保存了每个男性对 n 个女性偏好顺序（rank），按降序排列。数组 women 保存了女性的偏好。首先，数组 women 变换为数组 rang，保存每个女性 j 对每个男性 i 的偏好次序。例如，假如 rank[*j*][*i*] = 0，那么男性 i 是女性 j 最喜爱的选择，而 rank[*j*][*i′*] = 1 则意味着 i′ 是 j 的第二选择，以此类推。

最终，数组 husband 保存了所有女性被匹配的男性。而 unmarried 保存了仍是单身状态的男性列表。对于每个男性 i，next[*i*] 表示其偏好列表中下一个尝试匹配的女性编号。

```
def gale_shapley(men, women):
    n = len(men)
    assert n == len(women)
    next = [0] * n
    husband = [None] * n
    rank = [[0] * n for j in range(n)]        # 建立偏好排序 rank
    for j in range(n):
        for r in range(n):
            rank[j][women[j][r]] = r
    unmarried = deque(range(n))               # 所有男性都是单身
    while unmarried:                          # 当仍然存在未匹配男性的时候
        i = unmarried.popleft()
        j = men[i][next[i]]
        next[i] += 1
        if husband[j] is None:
            husband[j] = i
        elif rank[j][husband[j]] < rank[j][i]:
            unmarried.append(i)
        else:
            unmarried.put(husband[j])         # 对不起，husband[j] 被解除匹配
            husband[j] = i
    return husband
```

9.5　Ford-Fulkerson 最大流算法



- **应用**

寻找飞机乘客疏散路线中的瓶颈或者电网中的电阻，很多问题都可以用图的最大流问题来建模。

- **定义**

给定一个有向图 G(V, A)，并在弧上标注容量 c：A->R⁺，选择两个独立顶点：一个源点 $s\in$V 和一个汇点 $t\in$V。为了保持普适性，假设对于所有弧 (u, v)，弧 (v, u) 也在图中，因为我们总能在图中添加容量为 0 的弧，而不改变最优解。一个流是一个函数 f: A->R，它满足以下条件：

$$\forall(u,v)\in A: f(u,v)=-f(v,u) \tag{9.2}$$

对于所有容量值，它满足：

$$\forall e\in A: f(e)\leqslant c(e) \tag{9.3}$$

同样，把流转换为顶点（除了源点和汇点），有：

$$\forall v\in V\setminus\{s,t\}: \sum_{u:(u,v)\in A} f(u,v)=0 \tag{9.4}$$

流的值是离开源点的容量值，$\sum_u f(s,v)$ 。目的是找到一个值最大的流。

对于无向图，我们把每条边 (u, v) 替换成两条容量一致的边 (u, v) 和 (v, u)。

- **剩余图和增广路径**

对于一个给定的 f，我们考虑一个由剩余容量 $c(u, v)-f(u, v)$ 为正值的所有弧 (u, v) 组成的剩余图。在图中，一条从 s 到 t 的路径被称作增广路径，因为我们可以沿着这条路径来扩展流，扩展值 Δ 是路径 P 的弧的剩余容量最小值。为了在扩展 $f(u, v)$ 时保证 (9.2) 成立，必须把 $f(u, v)$ 减小同样的值。

- **复杂度为 $O(|V| \cdot |A| \cdot |C|)$ 的 Ford–Fulkerson 算法**

其中 $C = \max_{a \in A} c(a)$。为了让算法能够终止，所有容量必须是整数。使用循环方法，算法会寻找一条增广路径 P 并沿着 P 扩展流。通过深度优先遍历找到增广路径，我们仅能保证在每次遍历过程中，流会严格增长。流的值会明显增长 nC，最终得到所需复杂度。

- **实现细节**

在我们介绍的实现中，图以邻接数组的方式给出。但为了简化对流的操作，我们用一个二维矩阵 F 来表示流。Augment 方法尝试沿着通向汇点的路径，借助一个最大值 `val` 来扩展流。如果成功，Augment 方法返回流增加的值；如果失败，则返回 0。本方法通过深度优先遍历找到这条路径，并用 `visit` 标记。

```
def _augment(graph, capacity, flow, val, u, target, visit):
    visit[u] = True
    if u == target:
        return val
    for v in graph[u]:
        cuv = capacity[u][v]
        if not visit[v] and cuv > flow[u][v]:       # 可通过的弧
            res = min(val, cuv - flow[u][v])
            delta = _augment(graph, capacity, flow, res, v, target, visit)
            if delta > 0:
                flow[u][v] += delta                 # 扩展流
                flow[v][u] -= delta
                return delta
    return 0

def ford_fulkerson(graph, capacity, s, t):
    add_reverse_arcs(graph, capacity)
    n = len(graph)
    flow = [[0] * n for _ in range(n)]
    INF = float('inf')
    while _augment(graph, capacity, flow, INF, s, t, [False] * n) > 0:
            pass                                    # 空的循环体
    return(flow, sum(flow[s]))                      # 流和流的值
```

- **以二进制阻塞流算法进行优化的复杂度为 $O(|V| \cdot |A| \cdot \log C)$（见参考文献 [12]）**

一个可能的优化算法是，不再简单地扩展第一条已找到的路径，而是每次扩展一个较大的值。具体来讲，设图中边上的最大容量为 C；Δ 最大不超过 C 的 2 幂次（如果 C 是 9，那么 Δ 是 8；C 是 21，Δ 是 16；C 是 32，Δ 也是 32）。我们尝试循环使用容量至少是 Δ 的增广路径来扩展流。当这个操作无法实现时，我们可以尝试使用相当于二分之一 Δ 值的剩余容量值来扩展路径。当剩余图中的 s 和 t 都断开连接时，$\Delta = 1$ 的最终状态结束，这时算法一定能计算出一个最大流。

在最初状态中，根据 C 的定义，我们知道最大流的上限值是 $|V| \cdot C$。因此，最初状态最多只能

找到 n 条增广路径。找到一条增广路径的复杂度为 $O(|V|+|A|)$。由于只存在 $\log_2 C$ 个状态[①],算法的总复杂度就成了 $O(|V| \cdot |A| \cdot \log C)$。

9.6　Edmonds-Karp 算法的最大流

- **分析结论**

Ford-Fulkerson 算法不是多项式时间内解决问题的算法，它与输入数据量大小呈指数关系。幸好有一个转换方法，能让其复杂度成为多项式时间复杂度，并独立于 C。

令人吃惊的是，当我们应用最短增广路径时，同样算法的复杂度与最大容量无关。思路是最短增广路径的长度随着每次对 $|E|$ 的迭代严格递增。我们由此得到以下结论。

— 设分层图 L_f，其中 s 在第 0 层，所有从 s 出发且经过一条**不饱和**弧[②]到达的顶点为第一层的顶点，以此类推。因此，这是一个剩余图的无环子图。

— 在剩余图中，一条从 s 到 t 的最短路径一定是分层图中的一条路径。当我们沿着这条路径扩展流时，其中一条弧会变成饱和弧。

— 沿着一条路径的一次扩展会让某些弧变得不饱和，但仅是那些通向更低层的弧。因此，那些变成可通过状态的弧不能减少从 s 到 v 的路径长度（这一长度用弧的数量计算）。相应地，从 v 到 t 的路径长度也无法减少。

— 当执行 $|E|+1$ 次迭代后，一条弧 (u, v) 及其逆向弧 (v, u) 变得饱和。这证明了顶点 u 随时间会改变所在层次。通过前面的结论可以发现，从 s 到 t 的距离是严格增长的。

— 由于只存在 n 层，因而总共只会有 $|V| \cdot |E|$ 次迭代。

— 寻找最短路径通过时间复杂度为 $O(|V|)$ 的广度优先算法实现，总时间复杂度是 $O(|V| \cdot |E|^2)$。

时间复杂度为 $O(|V| \cdot |E|^2)$ 的 Edmonds–Karp 算法：从一个空的流开始，只要存在增广路径，就沿着最短增广路径进行拓展。

- **实现细节**

在一个队列 Q 的帮助下，广度优先算法能找到最短增广路径。数组 P 有两个作用。一方面，它用于标注已被广度优先算法遍历过的顶点[③]。在这种情况下，我们没有简单标注“是”或“否”，而是在内存中保存从路径源点到相关顶点的路径。另一方面，我们借此跟着 P 值回溯到源点，从而沿着路径来扩展流。对于每个已经访问过的顶点 v，数组 A 保存沿着从源点到 v 路径的最小剩余容量值。使用数组 A，我们可以确定沿着这条增广路径能把这个流扩展到多大。

---

① 由于 Δ 值不超过 C 的 2 次幂，且每次都会减半，因而状态数量一定是 $\log_2 C$。——译者注

② 当流达到了弧的容量的时候弧是饱和弧。

③ 想一想，为什么要标注 P[source]？

```
def _augment(graph, capacity, flow, source, target):
    n = len(graph)
    A = [0] * n                       # A[v]= 从源点到 v 的路径的最小剩余容量
    augm_path = [None] * n            # None = 尚未访问过的顶点
    Q = deque()                       # 广度优先遍历
    Q.append(source)
    augm_path[source] = source
    A[source] = float('inf')
    while Q:
        u = Q.popleft()
        for v in graph[u]:
            cuv = capacity[u][v]
            residual = cuv - flow[u][v]
            if residual > 0 and augm_path[v] is None:
                augm_path[v] = u      # 储存遍历过的点
                A[v] = min(A[u], residual)
                if v == target:
                    break
                else:
                    Q.append(v)
    return(augm_path, A[target])      # 增广路径，最小剩余容量

def edmonds_karp(graph, capacity, source, target):
    add_reverse_arcs(graph, capacity)
    V = range(len(graph))
    flow = [[0 for v in V] for u in V]
    while True:
        augm_path, delta = _augment(graph, capacity, flow, source, target)
        if delta == 0:
            break
        v = target                    # 回溯回源点
        while v != source:
            u = augm_path[v]          # 扩展流
            flow[u][v] += delta
            flow[v][u] -= delta
            v = u
    return(flow, sum(flow[source]))   # 流，流的值
```

9.7 Dinic 最大流算法

- **复杂度为 $O(|V|^2 \cdot |E|)$ 的 Dinic 算法**

Dinic 算法与前述算法是同时间被找到的。这次，我们没有在一个增广路径的集合中一个个地搜索，直到 s 和 t 之间距离增加，而是使用唯一一次遍历就找到这样一个流。算法复杂度变成了 $O(|V|^2 \cdot |E|)$。

设想一个函数 `dinic(u, val)`，它试图在一个分层图中让一个流从 u 到 t 通过。限制是这个

流不能超过 val。函数返回这个流的值，通过调用 u 的顺序，沿着从 s 到 u 的路径扩展流。具体来讲，为了把一个值为 val 的流从 u **推动**到 t，我们在分层图中遍历所有 u 的相邻顶点 v，并用递归方式让最大流从 v 通向 t。流的总和就是能把 u 推动到 t 的最大流。

函数 dinic(u, val) 会检测顶点 t 是否无法从 u 到达，从 u 到 t 是否不再有任何流。如果是，函数会从分层图中去掉 u，只需简单把它设定为“不存在”的 -1 级别。这样一来，随后的调用不必尝试从 u 通过一个流。很明显，在 $O(n)$ 次迭代后，s 和 t 会断开连接，然后我们重新计算一个新的分层图（图 9.5）。

即便使用一个邻接数组来表示图，用矩阵来表示流的剩余容量也非常有效。要注意，每个操作都满足对称性 $f(u, v) = -f(v, u)$。

```
def dinic(graph, capacity, source, target):
    assert source != target
    add_reverse_arcs(graph, capacity)
    Q = deque()
    total = 0
    n = len(graph)
    flow = [[0] * n for u in range(n)]      # 初始状态的空流
    while True:                             # 当可以扩展的时候重复
        Q.appendleft(source)
        lev = [None] * n                    # 按层建立，不可到达的时候 =None
        lev[source] = 0                     # 使用广度优先遍历
        while Q:
            u = Q.pop()
            for v in graph[u]:
                if lev[v] is None and capacity[u][v] > flow[u][v]:
                    lev[v] = lev[u] + 1
                    Q.appendleft(v)

            if lev[target] is None:         # 当汇点无法到达的时候停止
                return flow, total          # UB = 上界
            UB = sum(capacity[source][v] for v in graph[source]) - total
            total += _dinic_step(graph, capacity, lev, flow, source, target, UB)
def _dinic_step(graph, capacity, lev, flow, u, target, limit):
    if limit <= 0:
        return 0
    if u == target:
        return limit
    val = 0
    for v in graph[u]:
        residuel = capacity[u][v] - flow[u][v]
        if lev[v] == lev[u] + 1 and residuel > 0:
            z = min(limit, residuel)
            aug = _dinic_step(graph, capacity, lev, flow, v, target, z)
            flow[u][v] += aug
            flow[v][u] -= aug
            val += aug
            limit -= aug
    if val == 0:
        lev[u] = None                       # 去掉无法到达的顶点
    return val
```

- **实现细节**

数组 lev 保存着一个顶点在分层图中的层级。当再也不能从源点抵达汇点时，这个图会被重新计算。`Dinic_step` 方法会尽可能多地把流从 *u* 推动到汇点，而且不超过给定的限制。为了实现这一点，它在分层图中把尽可能多的流推动到自己的相邻顶点，然后在 `val` 中汇总已推动过的流的数量。当没有任何一个流从源点离开时，顶点 *v* 就被设置在 `None` 层，从而从分层图中移除。

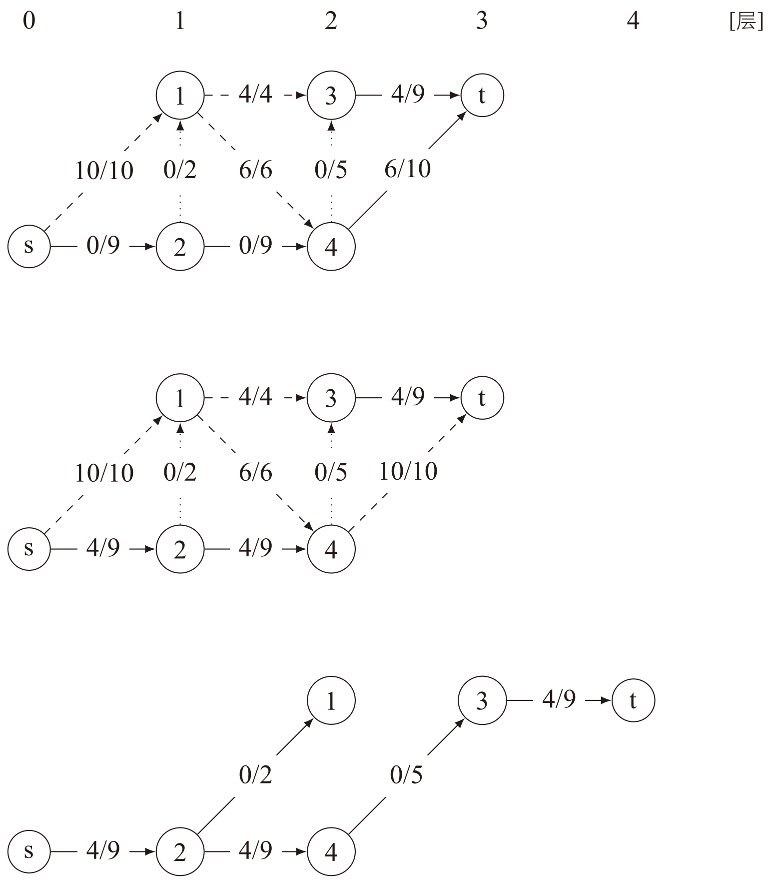


图 9.5　Dinic 算法过程中的分层图演化。图中仅展示了通向下一层的弧，弧用 ***f/c*** 来标注，其中 ***f*** 是流，***c*** 是容量。无法抵达的饱和弧用截线表示，即将成为分层图一部分的弧用点线表示。在沿着路径 ***s***-2-4-***t*** 把流扩展了 4 以后，顶点 ***t*** 被断开。这时必须重新计算分层图，重算后的分层图在最下方

9.8　*s-t* 最小割

- **应用**

在敌国，城市之间以道路连接，摧毁每条道路都需要一些成本。给定两个城市 *s* 和 *t*，目的是以最小成本来断开从 *s* 到 *t* 的道路连接。

- **定义**

问题的一个例子是包含两个不同顶点 *s* 和 *t* 的有向图 G(V, A)，每条弧 c 的成本是 $A\text{->}R^+$。*s-t* 割就是一个集合 S∈V，包含 *s* 但不包含 *t*。同样，割 S 有时以离开 S 的弧来定义，即那些满足 *v*∈S 和 *v*∈S 的弧 (*u*, *v*)（图 9.6）。割的值是这些弧的总成本。算法旨在找到一个成本最小的割。

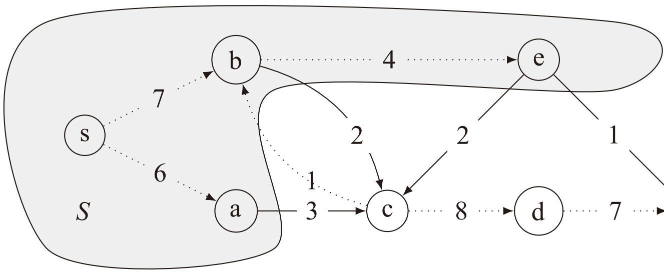


图 9.6　*s-t* 的一个最小割 S，成本为 3 + 2 + 2 + 1 = 8。离开 S 的弧用实线表示

- **与最大流问题的关系**

我们用一个最小割问题的成本 *c* 来识别一个最大流问题的容量 *c*。所有流必须通过这个割，因此所有割的值都是所有流的值的上界。但两者还存在如下更强的关系：

最大流最小割（Max–flow min–cut）定理指出，最大流的值与最小割的值相等。

这一定理在 1956 年由 Elias、Feinstein 和 Shannon 证明了一部分，由 Ford 和 Fulkerson 证明了另一部分。证明由一系列简单测试组成。

1. 对于一个离开割 S 的流 *f*，如 $f(S)=\sum_{u\in s,v\notin s} f(u,v)$，*f*(S) 的数量对所有 S 都一样。证明方式很简单，因为对于所有 *w*∉S，*w* ≠ *t*，先后通过 (9.4) 和 (9.2)，有：

$$
\begin{aligned}
f(S) &= \sum_{u\in S,v\notin S} f(u,v) = \sum_{u\in S,v\notin S} f(u,v) + \sum_{v} f(w,v) \\
&= \sum_{u\in S,v\notin S,v\neq w} f(u,v) + \sum_{u\in S} f(u,w) + \sum_{u\in S} f(w,v) + \sum_{u\notin S} f(w,v) \\
&= \sum_{u\in S\cup w,v\notin S\cup w} f(u,v) = f(S\cup w)
\end{aligned}
$$

2. 通过 (9.3) 有 *f*(*S*) ≤ *c*(*S*)，即离开 S 的流永远不会比 S 的值大。这证明了理论的一半，即最大流的最大值是最小割的值。

3. 现在，如果给定一个流 f，对于所有割 S 都有 $f(S) < c(S)$，那么存在一条增广路径，只需设定 $S = s$ 且 $P = \varnothing$ 。由于 $f(S) < c(S)$，因而存在一条边 (u, v) 满足 $u \in S$ 和 $v \notin S$ ，且 $f(u, v) < c(u, v)$。我们把 (u, v) 添加到 P 中。如果 $v = t$，那么 P 就是一条增广路径，否则把 v 添加到 S 中重新开始。

- **算法**

这就给出了解决最小割问题的一个算法。首先计算最大流；然后在剩余图中，通过剩余容量为正值的弧，可以确定从 s 出发能到达的顶点 v 的集合 S。这样一来，由于流有最优解就不该存在一条增广路径，因而 S 不包含 t。S 值最大化特性让所有离开它的弧都被流饱和，S 的剩余容量是 0，所以 S 是一个最小割。

9.9 平面图的 *s-t* 最小割

- **平面图**

当图是平面图 [1]，而且给定了平面的嵌入方式时，s-t 最小割问题能更有效地解决。为了简化描述，我们假设图是一个平面网格。网格由边连接起来的顶点组成，边把图切分成一个个蜂窝单元。设想网格是矩形的，源点在左下角，汇点在右上角。

- **双面图**

在一个双面图中，每个蜂窝单元都是一个顶点，且存在两个额外的顶点 s' 和 t'。顶点 s' 代表网格的左上角，t' 代表网格的右下角。如果两个顶点在原始图中被一条边分开，它们会在新图中被重新连接。两条边的权重是一样的（图 9.7）。

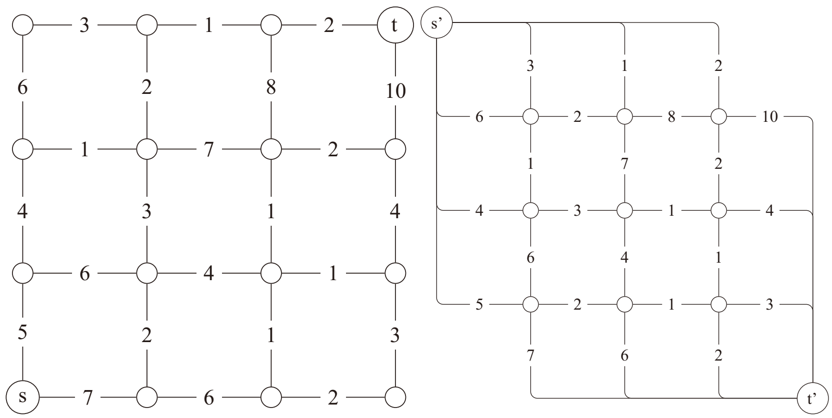


图 9.7　左边是原始图，右边是双面图：最短路 s'-t' 径对应着最小割 s-t

① 一个图若能在平面中描绘，而且任何边不出现交叉，它就是平面图。换句话说，当图中一个顶点与另一顶点相连，使得所有边（两点间线段）仅在顶点交叉时，图即为嵌入平面。——译者注

- **关键测试**

  在双面图中，所有长度为 w 的路径 s'-t' 对应着平面图中值为 w 的割 s-t，反之亦然。

- **复杂度为 $O((|V| + |E|)\log|V|)$ 的算法**

  为了计算这个矩阵中的最小割，只需使用 Dijkstra 这一类算法找到双面图中的最短路径即可。

- **变种**

  关键问题在于找到一个把图切断的最小割，也就是说，在所有顶点对 (s, t) 上的最小割 s-t。这个问题可以使用 Stoer-Wagner 算法在时间 $O(|V|\cdot|E|+|V|^2\log|V|)$ 内解决，它要比为每个顶点对 (s, t) 在时间 $\Theta(|V|^2)$ 内切割 s-t 更有意义（见参考文献 [25]）。

9.10　运输问题

流问题的一个推广问题是运输问题，其中存在多个源点和多个汇点。实际上，每个顶点 v 有一个值 d_v。当 d_v 为正，表示对外供应货物；当 d_v 为负，表示需要货物。问题必须满足 $\sum_{v\in V} d_v = 0$ 才能有解。目的是找到一个流，将运输量提供给需求方，因此这个流必须符合容量。而对于每个顶点 v，输入流与输出流的差值等于 d_v。在这此条件下，我们讨论的是“环流”而非“流”。通过添加源点和汇点，同时把源点与所有供应顶点连接，把汇点与所有需求顶点连接，这一问题可以很容易简化为流问题。

另一个变种问题把每个顶点与流的单位输送成本相关联。例如，假设一条弧 e 有一个流 $f_e = 3$ 和一个成本 $w_e = 4$，流在这条弧上造成的成本是 12。目的是找到一条让总成本最低的流。流可以是最大流，也可以是在运输问题中满足所有条件的流。因此，这一问题又称为**最低成本运输问题**。

为了解决问题，仍可以采用与 Kuhm-Munkres 算法（匈牙利算法）类似的算法。我们可以从最大流开始，然后沿着负成本环找到流，并扩展流。

9.11　在流和匹配之间化简

在二分最大匹配问题和最大流问题中，存在两个有趣的关系。

- **从匹配到流**

  首先，如果你有一个算法来计算一个图中的最大流，那么你就可以在一个二分图 G(U, V, E) 中计算最大匹配。为此，必须建立一个新图 G′(V′, E′)，其顶点 V′ = U ∪ V ∪ $\{s, t\}$ 包含了两个新顶点 s 和 t。源点 s 与 U 中的所有顶点连接，而 V 中任何顶点与汇点 t 连接。另外，每条边 (u, v)∈E 都与一条弧 (u, v)∈E′ 相关。G′ 中的所有弧都只有 1 个单位的容量。

  我们用多个层来表示这个图。第一层包含 s，第二层包含 U，第三层包含 V，最后一层包含 t。通过分层，G′ 中一个值为 k 的流即为 k 条穿越每个分层且不返回已经过层的路径。因此，由于容量

是 1 个单位的，顶点 $u \in U$ 最多只能被一条路径穿过，V 中其他顶点也一样。所以，流在 2 层和 3 层之间经过的边形成了大小为 k 的匹配。

- **从流到匹配**

另一方面，如果你有一个算法能够在一个二分图中计算出一个最大匹配，那么当容量为单位值时，你也能解决运输问题。设图 G(V, E) 且 $(d_v)_{v \in V}$ 是一个运输问题。

第一步，添加一个源点 s 和一个汇点 t。对于所有顶点 v，添加 $\max\{0, -d_v\}$ 条弧 (s, v) 和 $\max\{0, d_v\}$ 条弧 (v, t)。得到的结果是一个图 G′，在两个顶点之间会有多条连接弧，因此该图称为**多重图**。当且仅当新的多重图 G′ 有一个值为 $\Delta = \sum_{v:d_v>0} d_v$ 的流 s-t 时，最初的运输问题有解。

第二步，建立一个二分图 $H(V^+, V^-, E')$，它仅在 G' 有一个值为 Δ 的 s-t 流时满足完美匹配。

对于 G′ 中以 $e = (s, v)$ 形式表示的每条弧，生成一个顶点 $e^+ \in V^+$。对于 G' 中以 $e = (v, t)$ 形式表示的每条弧，生成一个顶点 $e^- \in V^-$。对于 G′ 其他的弧 e，生成两个顶点 $e^- \in V^-$ 和 $e^+ \in V^+$，并用一条边将它们连接。另外，对于所有弧 e、f，满足 e 的汇点与 f 的源点重合，生成一条边 (e^+, f^-)。

考虑 H 中的一个完美匹配。匹配中一条格式为 (e^-, e^+) 的边表示不存在穿过弧 e 的流。匹配中一条格式为 (e^+, f^-) 且 $e \neq f$ 的边表示一条穿过弧 e 然后穿过弧 f 的单位容量流。根据这一结构，所有连接源点或汇点的邻接弧都被流穿过。匹配即为一个值为 Δ 的 s-t 流（图 9.8）。

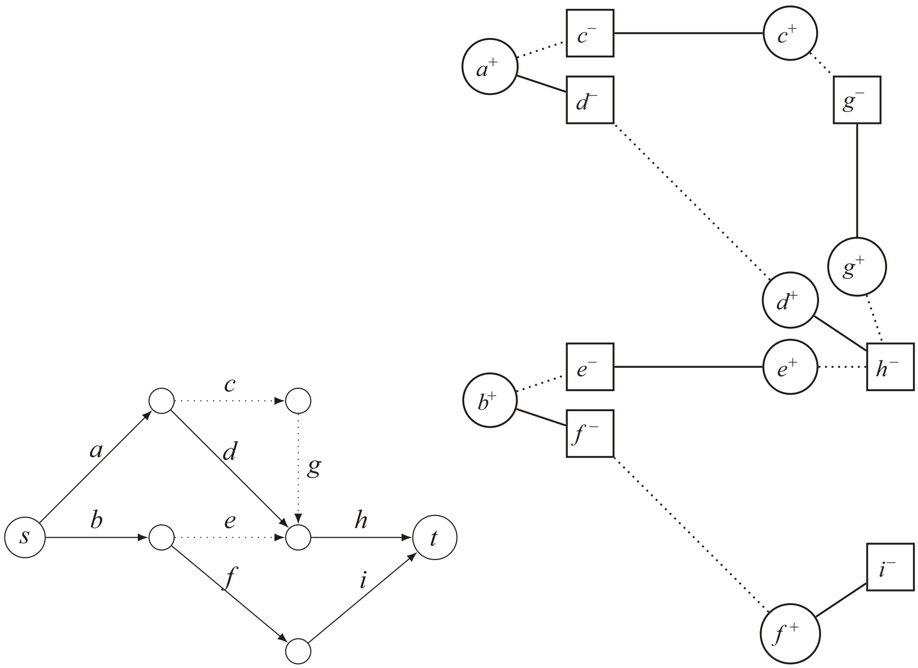
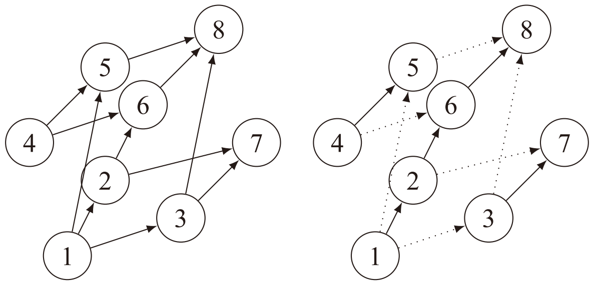


图 9.8　将单位容量的运输问题化简为二分最大匹配问题

9.12　偏序的宽度：Dilworth 算法



输入　　　　输出

- **定义**

给定一个偏序集，即一个无环、有向且有传递性的图 G(V, A)。G 中的链是一条有向路径，可引申为链是路径覆盖的所有顶点[①]。G 中的反链是一个顶点集合 S，不存在 $(u, v)\in$S，其中 $(u, v)\in$A。偏序的宽度指的是最长的反链长度。问题是如何计算这个宽度。

链和反链之间的重要关系由以下定理给出：

Dilworth 定理（1950）：**设最大反链的大小为** a**；把顶点拆分成为链，设链的最少数量为** b**，则有** $a = b$。[②]

对于一个反链 A，将它拆分后的链的集合为 B。链 A 与集合 B 中的每个链最多仅交叉于一个顶点。但是，每个顶点 $v\in$A 必属于 B 中的一条路径，因为 B 把图分成多个部分。故 $|A| \leqslant |B|$。Dilworth 定理证明了这两个问题的极限值相等。

为证明这一点，考虑一个二分图 H(V$^-$, V$^+$, E)，对于每个顶点 $v\in$V，存在一个顶点 $v^-\in$V$^-$ 且 $v^+\in$V$^+$，而对于每条弧 $(u, v)\in$A 存在边 $(u^-, v^+)\in$E。设 M 是图 H 的一个最大匹配。根据 Konig 定理（见 9.1 节），存在一个顶点集合 S，包含 H 中每条边的至少一个端点，且 $|M| = |S|$。

M 对应着图 G 被 B 中路径拆分的一个分区，它由边 (u, v) 组成，且满足 $(u^-,v^+)\in$M。所有路径都结束于 v，且 v^- 在 M 中是自由顶点，因此 $|B| = |V|-|M|$。

S 对应着一个反链 A，它由顶点 v 的集合组成，且满足 S 中不存在 v^- 或 v^+。由于 H 中每条边至少有一个端点在 S 中，因而没有任何一条弧的端点在 A 中，因此 A 是一条反链。

A 的大小至少是 $|V|-|S|$，因此 $|A| \geqslant |B|$。但是，由于反链的大小是链拆分后的分区大小的下限，由此可知二者相等。

---

① 链更常见的定义是一个图的顶点的子集，其中任意两个元素都可以比较。——译者注

② 链的最少划分数等于反链的最长长度。——译者注

- **复杂度为 $O(|V| \cdot |E|)$ 的算法**

  这个问题可化简为最大匹配问题（图 9.9）。

  — 建立一个二分图 $H(V^-, V^+, E)$，其中 V^- 和 V^+ 是 V 的副本；当 $(u, v) \in A$ 时，有 $(u^-, v^+) \in E$。

  — 在 H 中计算一个最大匹配 M。

  — 计算 U 中未匹配的顶点数量，这既是 G 中最大反链的大小，也是偏序 G 的大小。

  — 设 G 中弧的集合 D，且当 $(u, v) \in M$ 时，使得 $(u, v) \in D$。那么 D 是 V 被拆分后链数量最小的一部分。

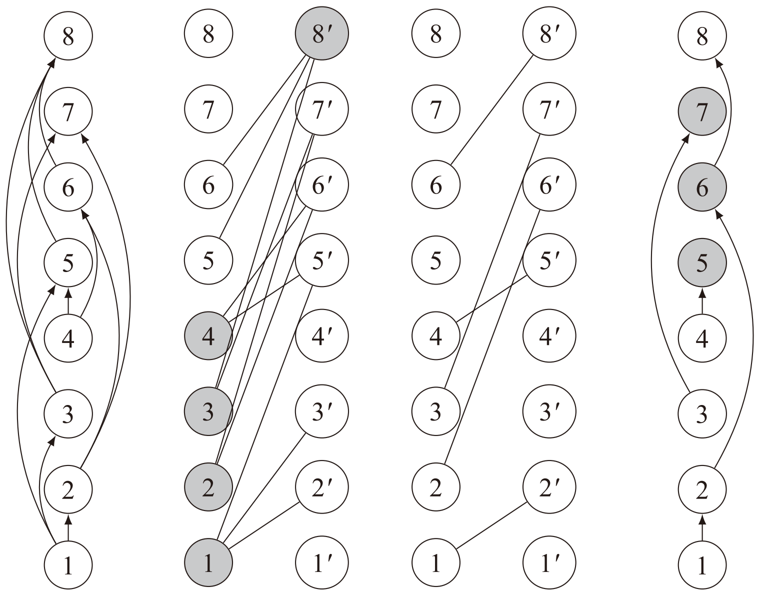


图 9.9　从左到右：一个偏序 G；相关二分图 H；H 中顶点 S 的覆盖用灰色表示；H 的一个最大匹配；G 中最大链数量的拆分，相关反链标注为灰色

- **预订出租车**

假设一家出租车公司拥有第二天的全部预约路线。公司必须在头天夜里把车队中的出租车分配给各个预约，并要让出租车的用量最小化。每个预约准确对应着一个确定的出发时间和出发地点，并去往另一个地方。假设所有路程的时间都是已知的，那么，我们可以确定同一辆出租车是否能在完成预约 i 后，继续完成预约 j。我们用 $i \leqslant j$ 表示这种情况。$\leqslant$ 关系是给定预约数据上的一个偏序，而问题解就是计算这个偏序的大小。

- **推广**

如果图有权重，我们可以寻找一个链的最小拆分，同时，这种拆分让所有弧的和最小化。这个问题可以化简为在二分图中寻找最小成本的完美匹配问题，并可在时间复杂度 $O(|V|^3)$ 内解决。

- **实现细节**

算法的实现返回一个数组 p，该数组编码了最优分区。$p[u]$ 是保存了顶点 u 的链的下标，链从 0 开始编码。函数的输入收到一个正方形矩阵 M，它对每个顶点对 u 和 v 标注了弧 (u, v) 的成本；当弧不存在时，成本值为 None。这个实现假设所有顶点都按拓扑顺序排序过，因此，如果弧 (u, v) 存在，那么有 $u<v$。

```
def dilworth(graph):
    n = len(graph)                                         # 最大割
    match = max_bipartite_matching(graph)                  # 链的分区
    part = [None] * n
    nb_chains = 0
    for v in range(n - 1, -1, -1):                         # 拓扑逆序
        if part[v] is None:                                # 链的开始
            u = v
            while u is not None:                           # 跟随链前进
                part[u] = nb_chains                        # 标注
                u = match[u]
            nb_chains += 1
    return part
```

第 10 章　树

树是一种组合数据结构，当我们考虑的对象建立在数据结构之上时，树就自然而然地出现了。其中最常见的考虑对象包括分类、层级关系、家谱等。一般来说，树结构需要用递归方式处理，算法的关键在于找到一个好的遍历方法。在这一章中，我们会见到很多树的经典问题。

正式来讲，树是一个联通无环图。树中一个顶点可以被指定为**根节点**，在这种情况下，树称作**有根树**。根节点给树中的弧赋予了父子关系。从一个顶点出发并沿着连接向上爬，就能抵达根节点。没有子节点的顶点被称作**叶子节点**。一个有 n 个顶点的树一定有 n-1 条边。为了证明这一点，我们从一棵树中轮流移除一个叶子节点和一条相邻边；在这个操作结束前，我们一定会得到一个孤立的顶点，树一个只有 1 个顶点和 0 条边。

基于树的动态数据结构有很多种，如红黑查找二叉树或线段树。这些结构实现了树的再平衡，以便让操作能够在对数时间内完成。相反在编程竞赛问题中，输入只给出一次，因此有时可以跳过这些操作，直接建立平衡的数据结构。

我们可以用两种方式来表达一个基本树。第一种是经典表示方式，即用邻接数组来描述，不区分特定的顶点，如树的根节点。另一种常用表示方式是前驱表方式：对于一个有根树（通常根节点是 0），除了根节点以外，每个顶点仅有一个唯一的前驱节点，后者被编码到一个表中。根据问题的类型和使用的算法，其中一种表示方式会更加匹配，而一种表示方式的树转换为另一种也可以在线性时间复杂度内完成。

```
def tree_prec_to_adj(prec, root=0):
    n = len(prec)
    graph = [[prec[u]] for u in range(n)]            # 添加前驱节点
    graph[root] = []
    for u in range(n):                               # 添加后序节点
        if u != root:
            graph[prec[u]].append(u)
    return graph
```

```
def tree_adj_to_prec(graph, root=0):
    n = len(graph)
    prec = [None] * len(graph)
    prec[root] = root                          # 标注，为了不重复访问根节点
    to_visit = [root]
    while to_visit:                            # 深度优先遍历
        node = to_visit.pop()
        for neighbor in graph[node]:
            if prec[neighbor] is None:
                prec[neighbor] = node
                to_visit.append(neighbor)
    prec[root] = None                          # 用标准方式标注根节点
    return prec
```

10.1　哈夫曼编码

- **定义**

一个字母表∑的二进制编码是一个函数 $c: \sum \rightarrow \{0,1\}^*$，且当 $a,b \in \sum$时，保证没有任何一个编码的字符 $c(a)$ 是另一个字符 $c(b)$ 的前缀。这个编码把∑中的每个字符变成了 $\{0,1\}^*$ 格式。而前缀上的特性可以清晰无误地解码出原始内容[①]。一般来说，我们希望编码尽可能地短。正式地说，给定一个频率函数 $f: \sum \rightarrow R^+$，我们寻找一个编码方式让以下式子最小，以便最小化成本：

$$\sum_{a \in \sum} f(a) \cdot |c(a)|$$

- **复杂度为 $O(n \log n)$ 的算法**

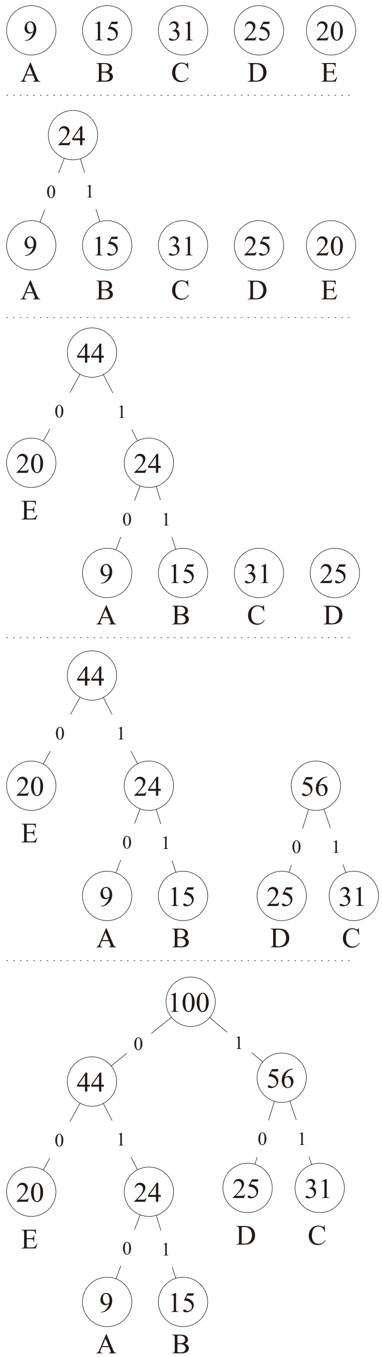
其中 n 是字母表的大小。哈夫曼编码可以被视为一棵二叉树，其叶子节点是字母表的每个字母，而每个节点用以该节点为根节点的子树的叶子节点的字符使用频率来标注。为了建立这棵二叉树，我们从一个森林开始，其中每个字母是一个单节点树，并以其使用频率标注。然后，当有多棵树时，我们把两个频率最低的树汇总在一起，再把这个新根节点用两棵子树的使用频率之和来标注（图 10.1）。通过参数交换，我们可以证明这样生成的编码是最优的。

为了能够操作这一树形结构，我们把它放置在一个能有效添加元素和删除最小元素的数据结构——优先级队列中。操作的时间成本与结构中对象的数量呈对数关系。一般我们使用堆来实现。在 Python 中，这一结构在 heapq 模块中。

存储在优先级队列中的元素是元组 (f_A, A)，其中 A 是一棵二叉树，f_A 是存储在 A 中的所有字符的使用频率之和。一棵树以两种方式来编码。一个字符 a 表示只有一个叶子节点（和根节点）的一

---

① 这里的编码格式用正则表达式来理解，{0, 1}* 即为包含且只包含多个或 0 个 0 和 1 字符的一个字符串，也就是二进制编码。——译者注



字符	输入频率	输出编码
A	7	00
B	7	01
C	7	10
D	7	11
A	40	1
B	5	01
C	2	001
D	1	000

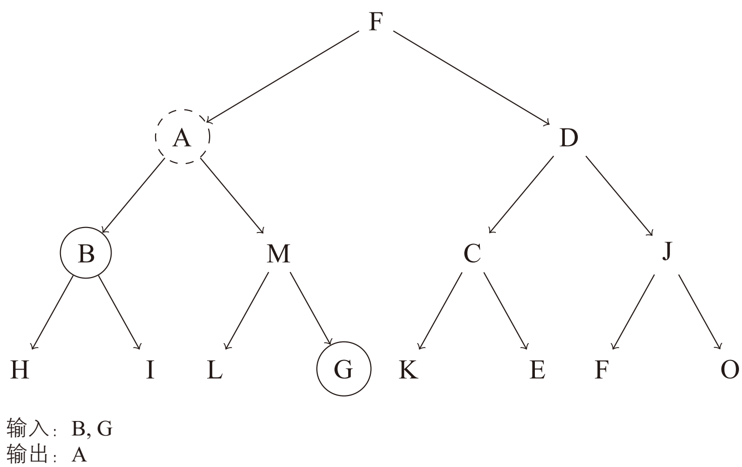
图 10.1　建立一个哈夫曼编码。每个节点被以其为根节点的子树的所有叶子节点的使用频率之和标注。在最下方，两个不同的输入产生了两个不同的哈夫曼编码

棵树。由左子树 l 和右子树 r 组成的一棵树用元组 (l, r) 表示，且它必须是一个列表[①]以避免重复。

```
def huffman(freq):
    h = []
    for a in freq:
        heappush(h, (freq[a], a))
    while len(h) > 1:
        (fl, l) = heappop(h)
        (fr, r) = heappop(h)
        heappush(h, (fl + fr, [l, r]))
    code = {}
    extract(code, h[0][1])
    return code

def extract(code, tree, prefix=""):
    if isinstance(tree, list):
        l, r = tree
        extract(code, l, prefix + "0")
        extract(code, r, prefix + "1")
    else:
        code[tree] = prefix
```

10.2　最近的共同祖先



- **定义**

给定一棵有 n 个节点的树，我们希望在对数时间内回复下面的查询：对于两给定顶点 u 和 v，找到它们在树中最近的**共同祖先**（lowest common ancestor，简称 LCA）。满足条件的节点 u' 把 u 和

① 必须是无重复元素的列表。——译者注

v 保存在 u' 的子树中，而且没有任何 u' 的直接后序节点有该属性。

- **每次查询复杂度为 $O(\log n)$ 的结构**

思路是为每个节点 u 添加一个层级信息和指向其祖先的引用，其中 anc[k, u] 是 u 的一个级别为 level[u]-2^k 的祖先节点值（如果祖先节点存在）；否则该值是 -1。因此，我们可以用这些“指针”快速追溯祖先节点的位置。

考虑查询 LCA(u, v)，即“谁是 u 和 v 的最近祖先?”在不失普适性的前提下，我们假设 level[u] ≤ level[v]。首先要选择与 u 在同一层级的 v 的祖先。然后对每个 k 从 $\log_2 n$ 到 0 迭代，如果 anc[k, u] ≠ anc[k, v]，那么用 u 和 v 的祖先节点 anc[k, u] 和 anc[k, u] 来替换它们。最终当 $u = v$ 时，我们找到了它们的共同祖先。

- **实现细节**

我们假设树以一个数组 prec 的形式给出，其中对于每棵树的节点 $u \in \{0, 1, \cdots, n-1\}$，且 prec[$u$] 表示父节点。当父节点下标比子节点下标小 1，且根节点是 0 时，以上假设成立。

```
class LowestCommonAncestorShortcuts:
    def __init__(self, prec):
        n = len(prec)
        self.level = [None] * n                        # 建立层级
        self.level[0] = 0
        for u in range(1, n):
                self.level[u] = 1 + self.level[prec[u]]
        depth = log2ceil(max(self.level[u] for u in range(n))) + 1
        self.anc = [[0] * n for _ in range(depth)]
        for u in range(n):
            self.anc[0][u] = prec[u]
        for k in range(1, depth):
            for u in range(n):
               self.anc[k][u] = self.anc[k - 1][self.anc[k - 1][u]]

    def query(self, u, v):
        # -- 假设 v 在树中没有 u 高
        if self.level[u] > self.level[v]:
            u, v = v, u
        # -- 让 v 与 u 在同级
        depth = len(self.anc)
        for k in range(depth -1, -1, -1):
            if self.level[u] <= self.level[v] - (1 << k):
                v = self.anc[k][v]
        assert self.level[u] == self.level[v]
        if u == v:
            return u
        # -- 升至最近的共同祖先
        for k in range(depth -1, -1, -1):
            if self.anc[k][u] != self.anc[k][v]:
                u = self.anc[k][u]
                v = self.anc[k][v]
        assert self.anc[0][u] == self.anc[0][v]
        return self.anc[0][u]
```

- **在一个区间内取最小值的替代方案**

考虑一个对树的深度优先遍历，如图 10.2 所示。为简化起见，假设所有顶点使用以下编号方式：所有顶点的编号都大于其父节点的编号。我们在一个数组 t 中记录这个遍历过程，在第一次和最后一次遇到顶点 u，以及在每次向下朝子节点递归时，都记录这一顶点。我们用 $f[u]$ 来记录处理顶点 u 的结束时间。现在，数组 t 在 $f[u]$ 和 $f[v]$ 之间包含了所有在 u 和 v 之间遍历的中间节点。这个区间中的最小顶点就是 u 和 v 的最低层祖先。因此，只需在线性时间内生成对树的深度优先遍历数组 t，并使用一个分段树来回复查询即可（见 4.5 节）。建立这一结构需要的时间是 $O(n\log n)$，一次查询的时间复杂度是 $O(\log n)$。

- **实现细节**

算法实现用邻接列表的方式接收输入的图，而且不假设各顶点的编号方式。因此，`dfs_trace` 记录不仅包含顶点，也包含元组（深度和顶点）。由于输入有可能很大，深度优先遍历通过一个栈 `to_visit` 来递归地实现。数组 next 表示对于每个顶点，有多少个后序节点已被遍历过。

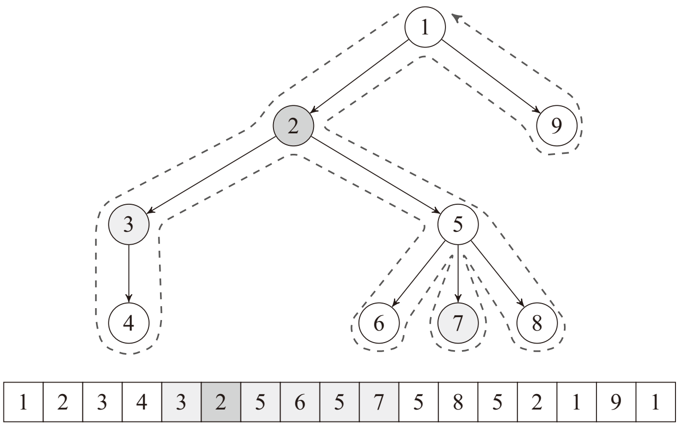


图 10.2　把最低层级共同祖先问题简化为在深度优先遍历过程中的区间最小值问题

```
class LowestCommonAncestorRMQ:
    def __init__(self, graph):
        n = len(graph)
        dfs_trace = []
        self.last = [None] * n
        to_visit = [(0, 0, None)]                          # 顶点 0 是树的根节点
        next = [0] * n
        while to_visit:
            level, node, father = to_visit[-1]
            self.last[node] = len(dfs_trace)
            dfs_trace.append((level, node))
            if next[node] < len(graph[node]) and \
               graph[node][next[node]] == father:
```

```
            next[node] += 1
        if next[node] == len(graph[node]):
            to_visit.pop()
        else:
            neighbor = graph[node][next[node]]
            next[node] += 1
            to_visit.append((level + 1, neighbor, node))
        self.rmq = RangeMinQuery(dfs_trace, (float('inf'), None))

    def query(self, u, v):
        lu = self.last[u]
        lv = self.last[v]
        if lu > lv:
           lu, lv = lv, lu
        return self.rmq.range_min(lu, lv + 1)[1]
```

- **变种**

使用这个数据结构，我们还可以确定树中两个节点之间的距离，因为通过最近共同祖先的路径一定最短。

10.3 树中的最长路径

定义：给定一棵树，寻找树中的最长路径。

复杂度：线性。

- **使用动态规划的算法**

和很多与树相关的问题一样，我们可以使用归纳子树的动态规划算法。固定一个根节点，以便把树中的边转向。

对于每个顶点 v，我们考虑一个以 v 为根节点的子树。用 $b[v]$ 来记录该子树中以 v 为终点的最长路径，用 $t[v]$ 来记录子树中没有限制条件的最长路径长度。我们也把 $b[v]$ 称作“以 v 为根节点的子树的深度”。

如果 v 没有子节点，则 $b[v] = t[v] = 0$。否则有以下关系：

$b[v] = 1 + \max b[u]$　　对于 v 节点的子节点 u

$t[v] = \max\{\max t[u_1], \max b[u_1] + 2 + b[u_2]\}$　　对于 v 节点的子节点 u_1 和 u_2

程序可以不必使用 −1 作为默认值来测试子节点数量。注意，没必要为了获得让 b 值最大化的两个子节点，而对节点的子节点进行排序。

- **陷阱**

如果树中有数百万个顶点，由于调用栈的大小限制①，使用 Python 进行深度优先遍历是无法实

① 在 Linux 系统下使用 ulimit -a 命令可以看到系统对栈的数量限制。——译者注

现算法的。因此，需要使用一个显式的栈[①]。

- **测试**

设一棵树中的随机顶点 r 和一个与 r 距离最远的顶点 u，于是在边界树 u 中存在一条最长路径。为了证明这一点，设一条端点为 v_1 和 v_2 的最长路径，在从 r 到 u 的路径上有顶点 u'，从 v_1 到 v_2 的路径上有顶点 v'，使得 u' 和 v' 的距离最小。如果这两条路径相交，我们可以在 u' 和 v' 的连接线中选择一个随机顶点（图 10.3）。

我们用 d 来记录树中的距离。对于 u，我们有[②]：

$$d(u', u) \geqslant d(u', v') + d(v', v_1)$$

由于 v_1 到 v_2 路径是最优的[③]，我们有：

$$d(v_1, v') \geqslant d(v', u') + d(u', u)$$

这使得 $d(u', v') = 0$ 且 $d(v', v_1) = d(v', u)$。因此，从 v_2 到 u 的路径同样也是树中一条最长路径。

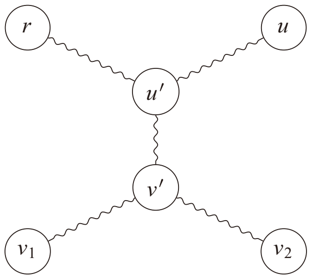


图 10.3　替换一个参数，证明当 u 是 r 的一个距离最远顶点时，存在一条从 u 出发的最长路径[④]

- **深度优先算法**

上述测试暗示存在一个替代算法。通过深度优先遍历，可以确定一个给定顶点的最远距离顶点。因此，我们可以选择一个随机顶点 r，确定一个距离 r 最远的顶点 v_1，然后重新确定一个距离 v_1 最远的顶点 v_2。从 v_1 到 v_2 的路径就是最长路径。

---

① 自己用程序去实现一个栈，而不是系统提供的栈。——译者注

② 因为从 r 到 u 的距离最远，因此在从 r 到 u' 一样的情况下，$u' \to v' \to v_1$ 的距离一定小于 $u' \to u$ 的距离。——译者注

③ 前面条件假设的存在一条端点为 v_1，v_2 的最长路径。——译者注

④ 记录 $d(u, u') = a, d(u', v') = b, d(v', v_1) = c$，我们会发现 $a \geqslant b + c$ [1]，且 $c \geqslant b + a$ [2]，那么把 [1] 中的 c 替换掉，得到 $a \geqslant b + b + a$，从而发现 $b = 0$。把 $d = 0$ 代入 [1] 和 [2] 得到 $a \geqslant c$ 和 $c \geqslant a$，从而得到 $a = c$。——译者注

- **变种**

我们希望在树中删掉尽可能少的边，使得由此产生的树中不存在长度超过 R 的路径。为此，只需在执行上述动态规划方法时，删掉确定下来的关键边即可。

考虑顺序处理顶点 v 及其子节点 $u_1,\cdots,u_d$，使得 $b[u_1]\leqslant\cdots\leqslant b[u_d]$。当 $b[u_d]+1>R$ 或 $b[u_{d-1}]+2+b[u_d]>R$ 时，删除边 (v, u_d)，然后减小 d，并重新开始测试。

10.4 最小权重生成树：Kruskal 算法

- **定义**

给定一个无向连通图，我们希望找到一个边的集合，使得所有顶点对能通过这些边连接起来。边的权重为正值，而我们想找的是权重和最小的边的集合。注意，在一个环中删除一条边仍能保留连通性，所以这种集合是无环的，它是一棵树——我们寻找的是一棵最小权重生成树（图 10.4）。

- **应用**

图中的边带着权重（或成本）w，我们要用最少的成本来添加边，最终让图连通，因此需要一个集合 $A\subset E$，使得 $G(V,A)$ 连通，且 $\sum_{e\in A}W(e)$ 值最小。

- **复杂度为 $O(|E|\log|E|)$ 的算法**

Kruskal 算法使用穷举法来解决问题，按照权重升序来遍历所有边 (u, v)，并在它们不能形成一个环的时候选择每条边 (u, v)。算法的最优性通过参数替换来证明。对于算法生成的解答 A 和一个任意解答 B，假设算法选择的第一条边为 e，且它不在 B 集合中，那么 $B\cup\{e\}$ 包含着一个环 C。通过选择 e，环 C 中所有边都有至少和 e 一样大的权重。因此，用 B 中的 e 替换其中一条边就能保留 B 的连通性。这样做不增加成本，仅仅减少了 A 到 B 之间的距离[①]。选择 B 作为越来越接近 A 的最优解，可以反证 A 就是最优解。

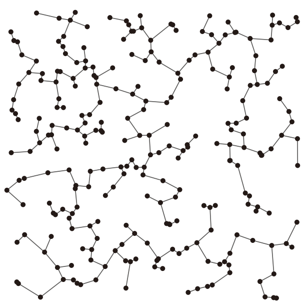


图 10.4　有 256 个顶点的完全图中的最小权重生成树，每条边的权重是它到其端点的欧氏距离

① 例如 A 与 B 之间的距离，我们选择 $|A\backslash B|+|B\backslash A|$。

- **实现细节**

为了能按照权重升序遍历所有边，我们要新建一个包含权重和边的数据对列表。列表根据数据对的字典序排序，并被遍历。为了维护森林，并能高效地检测加入一条新边后能否构成环，我们使用一个并查集结构（见 1.5.5 节）。

```
def kruskal(graph, weight):
    uf = UnionFind(len(graph))
    edges = []
    for u in range(len(graph)):
        for v in graph[u]:
            edges.append((weight[u][v], u, v))
    edges.sort()
    mst = []
    for w, u, v in edges:
        if uf.union(u, v):
            mst.append((u, v))
    return mst
```

- **Prim 算法**

问题有另外一种算法解法，即 Prim 算法，其运行方式和 Dijkstra 算法类似。Prim 算法为一个顶点集合 S 维护一个优先级队列 Q，其中包含着所有离开 S 的边。刚开始，S 包含唯一一个随机顶点 u；然后，只要 S 没有包含所有顶点，就从 Q 中取出一条权重最小的边 (u, v)，满足 $u \in S$ 且 $v \notin S$。顶点 u 被加入 S 中，Q 被更新。Prim 算法的复杂度也是 $O(|E|\log|E|)$。

第 11 章　集合

本章补充了序列的相关算法。其实，动态规划思想还能解决更多问题，就让我们从两个经典问题开始，即背包问题和找零问题。

11.1　背包问题

- **定义**

给定 n 个权重为 $p_0,\cdots,p_{n-1}$ 的对象，其各自对应的值为 $v_0,\cdots,v_{n-1}$，另设一个整数 C 作为背包的容量。我们希望知道如何得到一个对象值总和最大的子集，同时权重和不超过 C。这是一个 NP 复杂问题。

- **关键测试**

针对 $i\in\{0,\cdots,n-1\}$ 和 $c\in\{0,\cdots,C\}$，我们记可得的最大值为 Opt[i][c]，其中下标为 0 到 i 的对象权重和不超过 c（图 11.1）。对于基本情况 $i=0$，当 $p_0 > c$ 时，我们有 Opt[0][c] = 0，否则 Opt[0][c] = v_i。对于 i 取更大值，即 $i=1,\cdots,n-1$ 的情况，当对象下标为 i 时，最多有两种选择：要么选择它，要么不选择它。在第一种情况下，容量会减少 p_i，因此有如下关系：

$$\mathrm{Opt}[i][c]=\max\begin{cases}\mathrm{Opt}[i-1][c-p_i]+v_i & \text{在}c\geqslant c_i\text{时，取这个对象的情况}\\ \mathrm{Opt}[i-1][c] & \text{不取这个对象的情况}\end{cases}$$

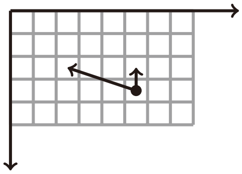


图 11.1　一个 Opt 表的展示。计算每个格子最多需要之前两个格子，其中包括一个位于正上方的格子。网格中的最长路径是问题的一个特殊情况，3.1 节有介绍

- **复杂度为 $O(nC)$ 的算法**

我们把有这种复杂度称作**伪多项式复杂度**。动态规划方法在维护矩阵 Opt 时，也要维护一个布尔型矩阵 Sel。后者记录下最终取得写入 Opt 的值的那一个选择。一旦这些矩阵依据上述递归方程被填满，对元素进行一次反向遍历，就能从 Sel 矩阵中找到取得最优解的元素集合。

```
def knapsack(p, v, cmax):
    n = len(p)
    Opt = [[0] * (cmax + 1) for _ in range(n + 1)]
    Sel = [[False] * (cmax + 1) for _ in range(n + 1)]
    #            -- 基本情况
    for cap in range(p[0], cmax + 1):
        Opt[0][cap] = v[0]
        Sel[0][cap] = True
    #            -- 归纳法
    for i in range(1, n):
        for cap in range(cmax + 1):
            if cap >= p[i] and Opt[i-1][cap - p[i]] + v[i] > Opt[i-1][cap]:
                Opt[i][cap] = Opt[i-1][cap - p[i]] + v[i]
                Sel[i][cap] = True
            else:
                Opt[i][cap] = Opt[i-1][cap]
                Sel[i][cap] = False
    #            -- 输出结果
    cap = cmax
    sol = []
    for i in range(n-1, -1, -1):
        if Sel[i][cap]:
            sol.append(i)
            cap -= p[i]
    return(Opt[n - 1][cmax], sol)
```

11.2　找零问题

现在，我们希望用面额为 $x_0,\cdots,x_{n-1}$ 分的硬币或钞票来获取一个值 R。问题在于确定是否存在一

个正值、线性组合 $x_0,\cdots,x_{n-1}$，其总和为 R。你或许会觉得可笑，但是缅甸就曾经使用面额为 15、25、35、45、75 和 90 缅元（kyat）的钞票（图 11.2）。

为了解决问题，一个值 x_i 可被多次用来获得一个总和值。

```
def coin_change(x, R):
    b = [False] * (R + 1)
    b[0] = True
    for xi in x:
        for s in range(xi, R + 1):
            b[s] |= b[s - xi]
    return b[R]
```

● **变种**

如果存在一个解决方案，那我们就可以尝试用最少数量的硬币或钞票解决问题。

● **测试**

只需按照货币面额降序计算，并随时保证后续选择的面额不超过剩余额度。



图 11.2　一张 45 缅元的旧钞票

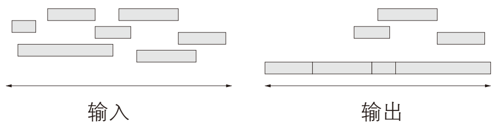
在欧元货币系统中，我们可以实现一个最小数量的货币组合。但是，假如货币面额体系为 1、3、4 和 10，而我们希望得到总和为 6。这个贪婪算法得到的最终结果是 3 个硬币的 4+1+1 组合，而不是最优解 3+3 组合。

● **复杂度为 $O(nR)$ 的算法**

假设货币面额是 $x_0,\cdots,x_{i-1}$，而需要的总额是 $0 \leqslant m \leqslant R$，则 A[$i$][$m$] 是所需货币数量最少的最终方案；当没有结果的时候，A[i][m] = ∞。我们可以派生出一个与背包问题类似的递归关系：对于所有额度 m，当 x_0 能把 m 整除时，A[0][m] 的值是 m/x_0，否则值是∞。当 $i = 1, \cdots, n-1$ 时，有如下关系：

$$A[i][m] = \max \begin{cases} A[i][m-x_i]+1 & \text{当}m \geqslant x_i\text{时，选择这一硬币的情况} \\ A[i-1][m] & \text{不选择这一硬币的情况} \end{cases}$$

11.3　给定总和值的子集



- **定义**

给定 n 个正整数 $x_0,\cdots,x_{n-1}$，我们希望知道是否存在一个整数和等于给定值 R 的子集。这是一个 NP 复杂问题。

- **复杂度为 $O(nR)$ 的算法**

维护一个布尔型数组，对于每个下标 i 和总和 $0 \leqslant s \leqslant R$，数组代表是否存在一个由整数 $x_0, x_1,\cdots,x_i$ 组成的子集，其总和等于 s。

起初，对于一个空集合，这一数组仅在下标等于 0 的位置是 `true`。然后，对于每个 $i\in\{0,\cdots,n-1\}$ 和所有 $s\in\{0,\cdots,R\}$，当且仅当存在一个总和为 s 或 $s-x_i$ 的子集 $x_0,\cdots,x_{i-1}$ 时，我们才能用整数 $x_0,\cdots,x_i$ 生成一个总和为 s 的子集。

注意代码实现中 s 上的循环执行顺序。

```python
def subset_sum(x, R):
    b = [False] * (R + 1)
    b[0] = True
    for xi in x:
        for s in range(R, xi - 1, -1):
            b[s] |= b[s - xi]
    return b[R]
```

- **复杂度为 $O(2^{\lceil n/2 \rceil})$ 的算法**

当 R 很大而 n 很小时，这个算法会很有趣。我们把输入 $X = \{x_0,\cdots,x_{n-1}\}$ 切分成两个不相交的部分 A 和 B，两部分最大为 $\lceil n/2 \rceil$。如果 S_A（S_B）是 A（B）每个子集元素的总和，我们建立一个集合 $Y = S_A$ 和集合 $Z = R - S_B$，两者包含着 R − v 数值对，其中 v 描述了 S_B。我们只需测试 Y 和 Z 是否有一个非空交集。

```python
def part_sum(x, i=0):
    if i == len(x):
        yield 0
    else:
        for s in part_sum(x, i + 1):
            yield s
            yield s + x[i]

def subset_sum(x, R):
```

```
    k = len(x) // 2                  # 切分输入
    Y = [v for v in part_sum(x[:k])]
    Z = [R - v for v in part_sum(x[k:])]
    Y.sort()                         # 测试 Y 和 Z 的交集
    Z.sort()
    i = 0
    j = 0
    while i < len(Y) and j < len(Z):
        if Y[i] == Z[j]:
            return True
        elif Y[i] < Z[j]:            # 增大最小元素的下标
            i += 1
        else:
            j += 1
    return False
```

- **变种：拆分成两个尽可能平衡的子集**

给定 $x_0,\cdots,x_{n-1}$，需要生成 $S \subseteq \{0,\cdots,n-1\}$，使得 $\left|\sum_{i\in S} x_i - \sum_{i\notin S} x_i\right|$ 尽可能小。

算法复杂度为 $O(n\sum x_i)$。如同处理部分总和一样，然后在布尔型数组 b 中寻找一个下标 s，使得 $b[s]$ 为 `true` 且最接近 $\sum x_i/2$。然后对于所有 $a=0,1,2,\cdots$ 和 $d=+1,-1$，考虑 $b[\sum x_i/2+a\cdot d]$。

11.4 *k* 个整数之和

- **定义**

给定 n 个整数 $x_0,\cdots,x_{n-1}$，我们希望知道能否从中取出 k 个数，使其总和为 0。

- **应用**

对于 $k=3$，这一问题在离散几何中非常重要，很多经典问题都能化简为求 3 个整数和的问题。例如，给定 n 个三角形，想知道它们能否完整覆盖另一个给定三角形。或者给定 n 个点，想知道是否存在一条直线通过其中至少 3 个点。对于这类问题，存在一个复杂度为 $O(n^2)$ 的算法，我们推测该方法是最优的。当 k 值更大时，问题在密码学中有着重要的应用价值。

- **复杂度为 $O(n^{k-1})$ 的算法**

首先测试 $k=2$ 的情况。只需测试是否存在 $i \neq j$ 且 $x_i=-x_j$。如果 x 是已排序的，使用一次双向遍历（如同合并两个有序列表）就能解决问题（见 4.1 节）。否则，可以把输入参数存入一个散列表，然后当 $-x_i$ 存在于表中时，在表中查找 x_i。

对于 $k=3$ 的情况，我们建议使用一个复杂度为 $O(n^2)$ 的算法。从给 x 排序开始；然后对每个 x_j 只需测试列表 $x+x_j$ 和 $-x$ 是否拥有一个公共元素，因为这个元素的格式一定是 x_i+x_j 和 $-x_k$，其中 $x_i+x_j+x_k=0$。这一方法在 k 值更大时能够进一步推广，但性能不如以下这种算法。

- **复杂度为 $O(n^{\lceil k/2 \rceil})$ 的算法**

通过求得 $k/2$ 个输入整数元素之和，我们建立一个给定整数的多重集 A[①]。同样，通过求得 $k/2$ 个输入整数元素之和，建立一个多重集 B。

现在，只需测试集合 A 和 R-B 是否有一个非空交集。为了实现这一点，我们把 A 和 B 排序，然后像合并两个有序列表一样，在两个列表上执行一次联合遍历。算法复杂度是 $O(n^{\lceil k/2 \rceil})$。

- **实现细节**

为避免多次取到同一个下标，我们在 A 和 B 中不仅存储总和，也存储由各个总和以及得到该总和的元素下标组成的数值对。因此，针对 A 和 B 中每个数值对，我们可以确定下标是否相交。

① 在多重集中，同一个元素可以出现多次。——译者注

第 12 章　点和多边形

几何问题的核心元素是点。点表示空间中的一个位置。本章介绍了很多与图上点相关的经典问题。

自然，我们会用坐标值对来表示点。另一个重要的基本操作就是测试方向（图 12.1）。给定三个点 a、b 和 c，我们希望知道这三个点是否排在一条直线上，或者是否有一条 $a \rightarrow b \rightarrow c$ 的前进路线，实现左转或右转。

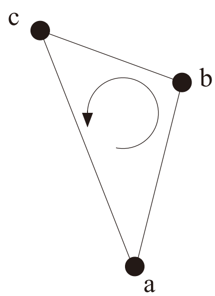
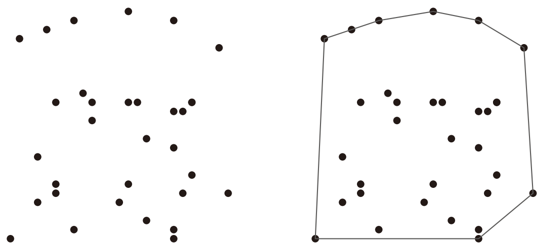


图 12.1　输入测试方法 **left_turn(a,b,c)** 会返回 **true**

```
def left_turn(a, b, c):
    return(a[0] - c[0]) * (b[1] - c[1]) - (a[1] - c[1]) * (b[0] - c[0]) > 0
```

当点坐标不是整数时，我们建议为了避免舍入误差，不要与 0 比较，而是与一个误差阈值进行比较，如 10^{-7}。

12.1　凸包问题



- **定义**

给定一个有 n 个点的集合，希望基于这些点的一个子集建立一个凸多边形，把剩余的点包含在多边形内。问题的解也是包含所有点且周长最小的多边形。

- **复杂度的下限**

通常情况下，不可能在时间 $o(n\log n)$ 内解决凸包问题①。为了证明这一点，我们设一个有 n 个数字的序列 $a_1,\cdots, a_n$。点 $(a_1, a_1^2),\cdots,(a_n, a_n^2)$ 的凸包计算的返回顺序与数字 $a_1,\cdots, a_n$ 的排序相关。因此，如果我们能在 $o(n\log n)$ 次操作内计算完毕，就会得到一个有同样复杂度的排序算法。

- **复杂度为 $O(n\log n)$ 的算法**

解决这个问题一般采用 Graham 扫描算法。但我们要介绍一个变种——Andrew 算法，它不会围绕着一个参考点去计算其周围点的角度，而是计算它们的 x 坐标。这一算法的好处在于不需要进行角度计算。角度计算经常带来精度差错。

我们仅介绍如何获取凸包的上部分。集合中的点会按照其 x 坐标升序来遍历，在一个 `top` 中，我们维护已被处理过的凸包的点。把每个新的点 p 加入 `top`；当倒数第二个点进入 `top` 使序列不再是凸多边形的时候，该点就会被移除。

- **实现细节**

凸包的下部分 `bot` 使用相同方式来获取。结果是把 `top` 列表反转，获得正确的凸包点顺序，即得到逆时针排序后的两个列表的拼接。注意，两个列表的第一个元素和最后一个元素相同，所以拼接结果中会出现重复，因此去掉一个多余的点非常重要。

为了简化代码，我们仅在删除了令序列不能形成凸多边形的元素后，再将点 p 添加入列表 `top` 和 `bot`。

① 这里是朗道表达式的小写 o。

```
def andrew(S):
    S.sort()
    top = []
    bot = []
    for p in S:
        while len(top) >= 2 and not left_turn(p, top[-1], top[-2]):
            top.pop()
        top.append(p)
        while len(bot) >= 2 and not left_turn(bot[-2], bot[-1], p):
            bot.pop()
        bot.append(p)
    return bot[:-1] + top[:0:-1]
```

12.2　多边形的测量

给定一个简单多边形 p[①]，格式为 n 个顺序正常[②]的点的列表形式，我们可以执行多个测量（图 12.2）。

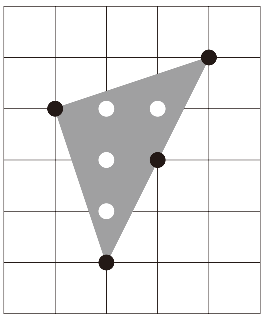


图 12.2　边缘上点（黑色）的数量是 4，内部整数坐标点（白色）也是 4 个，多边形面积是 4 + 4/2 - 1 = 5

- **在线性时间内计算面积**

我们可以使用以下公式计算 A 的面积：

$$A=\frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1}-x_{i+1}y_i)$$

其中下标 i+1 被除以 n 取模。第 i 个元素代表了三角形 $(0, p_i, p_{i+1})$ 带符号的面积，符号由三角形的方向决定。每个元素归结为计算少一个点的多边形面积。因此，多边形面积表示为三角形面积的加和减的序列。

① 当多边形的各个部分不相交时被称作简单多边形。

② 正常顺序既是逆时针方向。

```python
def area(p):
    A = 0
    for i in range(len(p)):
        A += p[i - 1][0] * p[i][1] - p[i][0] * p[i - 1][1]
    return A / 2.
```

- **计算边缘整数点的数量**

为了简化，我们假设多边形所有点的坐标都是整数。这样一来，针对每个 p 的分段 $[a, b]$，通过汇总在 a 和 b 之间点的数量来确定解。为了不重复计算，汇总不包括 a。如果 x 是 a 和 b 横坐标差的绝对值，y 是纵坐标差的绝对值，则点的数量是：

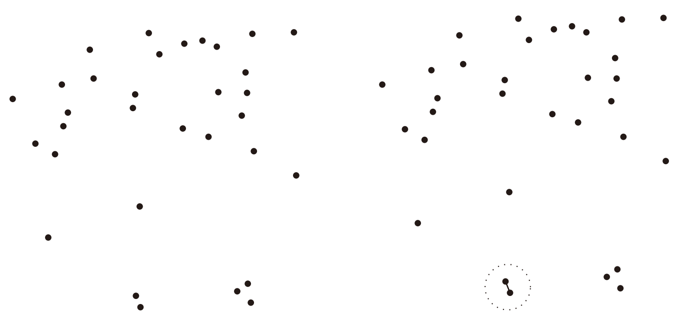
$$\begin{cases} y & \text{如果} x = 0 \\ x & \text{如果} y = 0 \\ x\text{和}y\text{的最大公约数} & \text{否则} \end{cases}$$

- **计算内部整数点的数量**

这个数量通过 Pick 定理获得，它与多边形的面积 A、内部整数点的数量 n_i 和边缘点的数量 n_b 有关，它们的关系由以下公式定义：

$$A = n_i + \frac{n_b}{2} - 1$$

12.3　最近点对



输入　　　　输出

- **应用**

宿营区随机摆放了很多帐篷。每个帐篷里都住着一个宿营者，拿着收音机。我们希望为所有宿营者限定一个音量值，好让任何人都不会被邻居的音乐声打扰。

- **定义**

给定 n 个点 $p_1,\cdots,p_n$，确定一个点对 (p_i,p_j)，使得 p_i 和 p_j 之间的欧氏距离最短。

- **线性时间复杂度的随机算法**

这一经典问题有好几个复杂度为 $O(n\log n)$ 的算法，使用的都是扫描法或分治法。我们下面介绍一个线性时间复杂度的随机算法，也就是说，期望计算时间是线性的。根据我们的经验，其效率只是略微优于扫描算法，但实现上更加简单。

思路是在任何情况下，我们已经找到了一个距离为 d 的点对，希望知道是否存在另一个距离更近的点对。为此，我们把空间在两个方向上分成步长[①] 为 $d/2$ 的网格。因此，每个点都属于网格中的一个格子。设已确定点对之间距离至少是 d 的所有点的集合为 P，那么每个格子最多包含一个 P 中的点。

网格由一个字典 G 表示，把每个格子及其包含的 P 中的点关联起来。当把 P 中的一个点 p 添加到 G 中时，只需测试点 p 与最近的 5×5 网格中点 q 的距离（图 12.3）。当发现一个点对的距离为 $d' < d$ 时，我们把网格步长设置为 $d'/2$，然后从头开始上述流程。

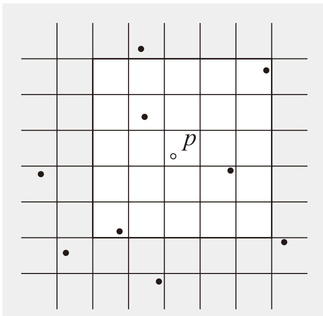


图 12.3　每个网格中的格子最多包含一个点。当考虑一个新的点 p 时，只需测量它与周围（白色）格子中点的距离即可

- **复杂度**

假设访问 G 的时间都是常数，那么计算包含给定点的格子的时间也是常数。关键论据是，假如给定输入的所有点都能按照统一的随机顺序来处理，那么在处理第 i 个点时 $(3\leqslant i\leqslant n)$，优化距离 d 的概率是 $1/(i-1)$。所以，期望复杂度数量级是 $\sum_{i=3}^{n} i/(i-1)$，与 n 呈线性关系。

- **实现细节**

为了在给定步长的网格中计算与点 (x, y) 关联的格子，只需把各个坐标除以步长，然后取整即可。特别要注意负值坐标，因为在 Python 和其他语言中，如 $-1/2$ 的取整结果是 0，而不是我们想要的 -1。

① 网格中一个格子的宽度。——译者注

最终，选择网格步长为 *d*/2 而不是 *d*，保证每个格子中只有一个元素，方便处理。

```
from math import hypot                  # hypot(dx, dy) = sqrt(dx * dx + dy * dy)
from random import shuffle

def dist(p, q):
    return hypot(p[0] - q[0], p[1] - q[1])

def floor(x, pas):
    return int(x / pas) - int(x < 0)

def cell(point, pas):
    x, y = point
    return(floor(x, pas), floor(y, pas))

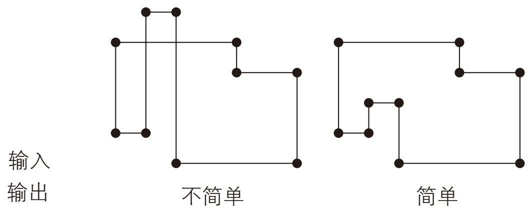
def ameliore(S, d):
    G = {}          # 网格
    for p in S:
        (a, b) = cell(p, d / 2.)
        for a1 in range(a - 2, a + 3):
            for b1 in range(b - 2, b + 3):
                if(a1, b1) in G:
                   q = G[a1, b1]
                   pq = dist(p, q)
                   if pq < d:
                       return(pq, p, q)
        G[a, b] = p
    return None

def closest_points(S):
    shuffle(S)
    assert len(S) >= 2
    p = S[0]
    q = S[1]
    d = dist(p, q)
    while d > 0:
        r = ameliore(S, d)
        if r:
            (d, p, q) = r
        else:
            break
    return(p, q)
```

12.4　简单直线多边形

- **定义**

当一个多边形的所有边在水平方向和垂直方向上交替切换时，它被称作**直线多边形**。当其所有边都不交叉时，它就是简单的。目的是测试一个给定的直线多边形是否简单。



- **测试**

如果一个多边形是直线的，那么其每个点之前和之后的边都有一个与之水平和垂直的边。因此，多边形的点之间可以是左右和上下关系，一个简单的相邻测试足以完成标注（图 12.4）①。

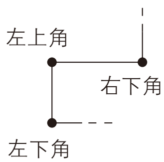


图 12.4 直线多边形中点的类型

- **复杂度为 $O(n\log n)$ 的算法**

算法通过扫描实现。按照坐标的字典序扫描多边形中的点，并维护一个还没有访问其右侧点的左侧点集合 S。对于每个点，S 刚开始是空值，我们执行下面的操作。

— 如果 (x, y) 与最后一个被处理的点相同，那么在多边形的点之间存在重叠。

— 如果 (x, y) 是左侧点，检查确认 y 尚未存在于 S 中：这意味着我们已经访问过同一纵坐标上且正在向右转的点，因此这两个横向元素是重叠的。

— 如果 (x, y) 是右侧点，那么 y 必须存在于 S 中，因为其左侧邻点已被访问过。这时，我们把 y 从 S 中剔除。

— 如果 (x, y) 是下方点，什么都不做。

— 如果 (x, y) 是上方点，那么设其下方邻点为 (x, y')。如果这不是刚刚被处理过的点，则说明线段 (x, y') – (x, y) 和另一条垂直线段是重叠的；否则，我们检查 S，查找满足 $y'<y''<y$ 的值 y''。这说明一条纵坐标为 y'' 的水平线段和当前垂直线段 (x, y') – (x, y) 交叉，因此多边形就不是简单的。

- **复杂度**

为了获得一个合理的复杂度，在 S 上执行的操作要足够高效，比如：

— 向 S 中添加和移除元素；

① 每条边都是水平、垂直切换，所以两条边一定会以 90° 角相交于一个点。这个点可类比成一个正四边形的 4 个角点，图 12.4 中就用左上、左下、右上、右下来对直线多边形中的点来进行分类。

——译者注

— 检查 S 是否包含一个给定区间 $[a, b]$ 内的元素。

如果我们用一个数组 t 来表示 S，使得当 $y\in S$ 时 $t[y]=-1$，否则 $t[y]=0$，那么确定在 S 中是否存在一个区间 $[a, b]$ 的操作就变成在 $t[a]$ 和 $t[b]$ 之间的区间中寻找 -1。因此，我们用一棵线段树来表示 t（见 4.5 节），从而让查询一个区间中最小值和更新数组 t 的操作能在对数时间内实现，保证算法复杂度是 $O(n\log n)$。

- **实现细节**

比起处理没有上下限的点的纵坐标，我们更关心点的次序。设所有满足 $k \leqslant n/2$ 的点的不重复纵坐标的列表为 $y_0<\cdots<y_k$，于是当且仅当 $t[i] = -1$ 时，有 $y_i\in S$。为了确定 S 在区间 $[y_i, y_k]$ 中包含一个元素 y_j，只需确定 t 在 $t[i+1]$ 和 $t[k-1]$ 之间的最小值是 -1。

```
def is_simple(polygon):
    n = len(polygon)
    order = list(range(n))
    order.sort(key=lambda i: polygon[i])      # 字典序
    rank_to_y = list(set(p[1] for p in polygon))
    rank_to_y.sort()
    y_to_rank = {rank_to_y[i]: i for i in range(len(rank_to_y))}
    S = RangeMinQuery([0] * len(rank_to_y))   # 扫描结构
    for i in order:
        x, y = polygon[i]
        rank = y_to_rank[y]
        #        -- 点的类型
        right_x = max(polygon[i - 1][0], polygon[(i + 1) % n][0])
        left = x < right_x
        below_y = min(polygon[i - 1][1], polygon[(i + 1) % n][1])
        high = y > below_y
        if left:                              # S 中不能有 y
            if S[rank]:
               return False                   # 两条水平线段相交
            S[rank] = -1                      # 把 y 添加入 S
        else:
            S[rank] = 0                       # 从 S 中删除 y
        if high:
            lo = y_to_rank[below_y]           # 确认 S 在 lo + 1 和 rank - 1 之间
            if(below_y != last_y or last_y == y or
               rank - lo >= 2 and S.range_min(lo + 1, rank)):
               return False                   # 垂直和水平线段交叉
        last_y = y                            # 记录下来，准备下一次迭代
    return True
```

第 13 章　长方形

很多处理几何图形的问题与长方形有关，比如房屋图纸或计算机屏幕的显示。长方形有时是直线多边形，即边和轴平行，这让处理变得容易。几何学中一个重要的算法技巧是扫描，在 13.5 节中有介绍。

13.1　组成长方形

- **定义**

给定图上 n 个点的集合 S，我们希望确定 S 中有 4 个角点的所有长方形。这些长方形不一定是直线多边形。

- **算法复杂度为 $O(n^2+m)$**

其中 m 是解的数量。关键测试要看两对对角点的签名是否一致。签名由中心以及点与点之间的距离组成。为测试签名是否一致，只需在一个字典中存储拥有相同键 (c, d) 的所有点对 p 和 q，其中 $c = (p+q)/2$ 是 p 和 q 的中心点，$d = |q - p|$ 是两点间距离。拥有相同签名的点对就是组成 S 中正方形的点（图 13.1）。

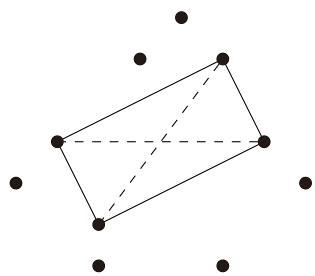


图 13.1　对角点对的相同签名，对角线从中被切开，从中间到各点距离相等

- **实现细节**

为了让算法的性能更好并只处理整数，在计算 c 时，除以 2 的操作被忽略，在计算 d 时，根被省略[①]。

```
def rectangles_from_points(S):
    answ = 0
    pairs = {}
    for j in range(len(S)):
        for i in range(j):
            px, py = S[i]
            qx, qy = S[j]
            center = (px + qx, py + qy)
            dist = (px - qx) * (px - qx) + (py - qy) * (py - qy)
            sign = (center, dist)
            if sign in pairs:
                answ += len(pairs[sign])
                pairs[sign].append((i, j))
            else:
                pairs[sign] = [(i, j)]
    return answ
```

13.2　网格中的最大正方形

- **定义**

给定一个格式为 $n \times m$ 的点阵黑白图像，我们希望确定其中最大的纯黑色方块（图 13.2）。

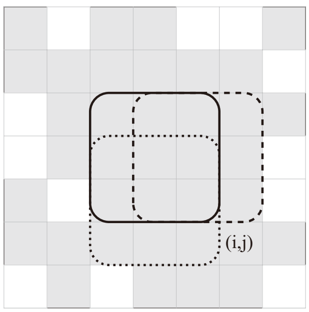


图 13.2　一个面积为 k 的最大黑色色块，其右下角点 (i, j) 包含三个大小为 k-1 且右下角点为 $(i, j$-1$)$、$(i$-1$, j$-1$)$ 和 $(i$-1$, j)$ 的正方形黑色色块

① 在计算两点间距时需要开平方根，因为 c 和 d 只作为签名使用而并不需要算出实际距离，因此不需要开平方。——译者注

● **线性时间复杂度的算法**

这个问题可以简单使用动态规划算法解决。假设所有行被从上到下编号，所有列被从左到右编号。一个方块的右下角点为 (i, j)，就称其“结束于 (i, j)”。如果这个方块的边长为 k，那么它由格子 (i', j') 组成，并有 $i-k < i' \leqslant i$ 且 $j - k < j' \leqslant j$。

对于网格的每个格子 (i, j)，我们寻找最大整数 k，使得边长为 k 且结束于 (i, j) 的正方形为纯黑色。将该值记作 $A[i, j]$。如果格子 (i, j) 是白色的，那么 $A[i, j] = 0$，说明这个正方形不存在。

所有边长为 k 的纯黑色正方形包含 4 个边长为 k-1 的正方形。因此，当且仅当 $A[i-1, j]$、$A[i-1, j-1]$ 和 $A[i, j-1]$ 都至少等于 k-1，且有 $k \geqslant 1$ 时，$A[i, j] = k$。由此可得以下递归公式：

$$A[i, j] = \begin{cases} 0 \\ 1 + \min\{A[i-1, j], A[i-1, j-1], A[i, j-1]\} \end{cases}$$

13.3　直方图中的最大长方形

● **定义**

给定一个直方图，其格式是由正整数或空值 $x_0, \cdots, x_{n-1}$ 组成的数组。目标是在这个直方图中找到一个面积最大的长方形。也就是说，找到一个区间 $[l, r)$，使得面积 $(r - l) \times h$ 最大且 $h = \min_{l \leqslant i < r} x_i$。

● **应用**

在大西洋底铺设着连接了欧洲和美洲的通信电缆。这些电缆的技术特性会因海水和温度的变化而随时间改变。因此，在任何时候都会有一个随时间变化的最大传输速率。在信号传输过程中可以改变传输速率，但在终端之间变化传输速率会影响期间的所有通信。假设我们在一天中每 60 × 24 分钟提前知道了最大通信速率。现在，我们希望找到一个时间区间和一个速率，以便传输最大量信息又不会断开连接。问题归结为在一个直方图中找到一个最大面积长方形的问题。

● **线性时间复杂度的算法**

这是一个扫描算法。对于每个数组的前缀 $x_0, \cdots, x_{i-1}$，维护一个长方形集合，而我们尚未确定长方形的右侧边。这些长方形通过一个整数对 (l, h) 定义，其中 l 是左边界，h 是高度。那么，我们只需考虑其中最大的长方形，这样一来，h 就是 $x_l, x_{l+1}, \cdots, x_{i-1}$ 中最大的，且在 $x_{l-1} < h$ 时有 $l = 0$。因此，这个长方形无法在不超出直方图的情况下向左或向上变大。我们把整数对存储到一个按照 h 排序的栈中。有意思的是，这些整数对同样也按照 l 排序。

现在，对于每个值 x_i，我们可能已经找到了某些长方形的右侧边。当 $h > x_i$ 时，在栈上以 (l, h) 编码的所有长方形也是这种情况。这样一个长方形的宽度是 $i - l$。但 x_i 的值同样会新建一个新整数对 (l', x_i)。左侧边 l' 要么是最后一个出栈的长方形 l 值；要么在没有出栈操作时，$l' = i$（图 13.3）。

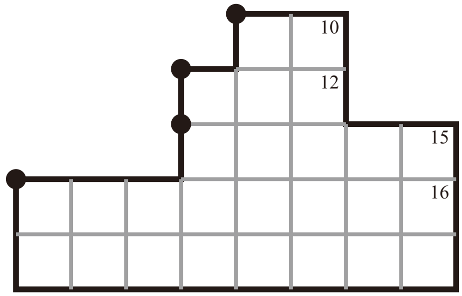


图 13.3　当直方图数字增长时，拐点就会堆叠起来；当直方图数字减小时，需要让过高的那些角点出栈。测试所有可能的长方形，再把最后一个出栈的角点补到更低的高度

```
def rectangles_from_histogram(H):
    best = (float('-inf'), 0, 0, 0)
    S = []
    H2 = H + [float('-inf')]          # 用额外的元素清空栈
    for right in range(len(H2)):
        x = H2[right]
        left = right
        while len(S) > 0 and S[-1][1] >= x:
            left, height = S.pop()
            #       ( 面积 , 左侧 , 高度 , 右侧 )
            rect = (height * (right - left), left, height, right)
            if rect > best:
                best = rect
        S.append((left, x))
    return best
```

13.4　网格中的最大长方形

- **应用**

给定一块布满了树的建筑工地，我们希望找到一块面积最大的长方形地块来建房子，同时不需要砍树。

- **定义**

给定一个格式为 $n \times m$ 的像素点阵黑白图片，我们希望确定其中最大的纯黑色长方形。这里的长方形是一段相交成行的网格和一段成列的网格（图 13.4）。

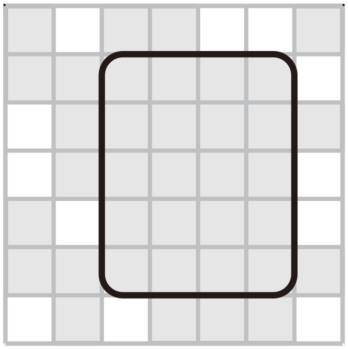


图 13.4　网格中面积最大的黑色长方形

- **线性时间复杂度的算法**

解决方案是把问题简化为查找一个直方图中的最大长方形问题。对于每行 i，寻找底部位于 i 行的最大长方形。为此，我们维护一个数组 t，它给每个列 j 一个最大数量 k，使得位于 (i, j) 和 $(i, j-k+1)$ 之间所有像素点都是黑色。因此，t 定义了一个直方图，我们在其中寻找最大的长方形。数组 t 根据每一列的像素颜色一行接一行地更新。

```
def rectangles_from_grid(P, noir=1):
    rows = len(P)
    cols = len(P[0])
    t = [0] * cols
    best = None
    for i in range(rows):
        for j in range(cols):
            if P[i][j] == noir:
                t[j] += 1
            else:
                t[j] = 0
        (area, left, height, right) = rectangles_from_histogram(t)
        alt = (area, left, i, right, i-height)
        if best is None or alt > best:
            best = alt
    return best
```

13.5　合并长方形

- **定义**

给定 n 个直线长方形，我们希望计算其并集的面积。使用同样的技术，我们可以计算其边长和连通区域的数量。

- **复杂度为 $O(n^4)$ 的算法**

无论何种算法，都要测试所有长方形的边缘在每个轴上是否最多有 $2n$ 个点。这些点形成一个网格，由 $O(n^2)$ 个格子组成。这些格子要么完全被一个长方形覆盖，要么与所有长方形分离（图 13.5）。第一个简单方案是确定一个布尔型数组，说明每个格子是否是长方形并集的一部分。只需计算格子的总面积就能知道并集的面积。处理每个长方形的工作时间是 $O(n^2)$，算法的整体复杂度为 $O(n^4)$。

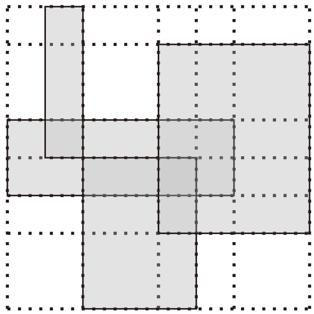


图 13.5　网格有 $O(n)$ 行和 $O(n)$ 列，每个格子要么在长方形的并集中，要么与之分离

- **复杂度为 $O(n^2)$ 的扫描算法**

我们用一条水平线从上到下扫描所有长方形。长方形的左侧边和右侧边把横轴分割为 $O(n)$ 个区间。在所有情况下，我们维护一个布尔型数组，它代表着每个区间被几个长方形覆盖。对至少被一个长方形覆盖的区间长度求和，我们可以确定扫描线与长方形并集的重叠长度。当我们把扫描线向下移动 Δ 单位时，只需把重叠长度乘以 Δ，并将其记入一个并集面积总数的变量中。

扫描线沿着变化事件一点点地前进。一个变化事件表示一个长方形的顶边或底边。处理一个顶边会增加长方形覆盖区间的计数器，而处理一个相关底边会减少计数器。这部分需要的时间成本是 $O(n)$，算法的整体复杂度即为 $O(n^2)$。

- **复杂度为 $O(n\log n)$ 的算法**

这里使用的数据结构——线段树，与在查询下标范围内最小值问题中使用的数据结构类似（见 4.5 节）。该数据结构被两个大小为 $2n-1$ 的数组 L 和 t 初始化。思路是长方形的横坐标把横坐标轴分割成了很多分段。第 i 条线段的长度是 $L[i]$。在从左往右扫描时，我们希望为每个分段维护包含该线段的长方形数量。这一信息被存储在数组 t 中。

数据结构可执行以下操作：

— `change(i, k, d)` 方法将值 d 添加到输入 $t[j]$ 中，且 $i \leqslant j<k$；

— `cover()` 方法返回在下标 j 上的总和$\sum L[j]$，其中 $t[j] \neq 0$。

`change` 操作在扫描遇到一个长方形的定边 ($d = 1$) 或底边 ($d = -1$) 时被调用。`cover` 方法用于

确定扫描线和长方形的重合线段长度。扫描线每次下降时，这一信息与扫描线的垂直移动距离相乘，我们可以借此确定长方形并集的面积。

数据结构由一棵二叉树组成，其每个节点 p 负责数组 t 中（也是 L 中）的一个下标区间 I。二叉树有三个属性：

— 当 $j \in$ I 时，$l[p]$ 是 $L[j]$ 的总和；

— 当 $j \in$ I 时，$c[p]$ 是被加入所有 $t[j]$ 的一个数；

— 当 $j \in$ I 且 $t[j] \neq 0$ 时，$s[p]$ 是 $L[j]$ 的总和。

树的根是 1 号节点，`cover()` 的结果被存在 $s[1]$ 中。数组 t 隐式地存储在属性 c 中。从下标 j 相关节点到根节点的路径上所有节点 p 的 $c[p]$ 值之和为 $t[j]$。与最小区间查询（minimum range query）结构一样，其更新需要呈对数的时间复杂度。

```
class Cover_query:
    def __init__(self, _len):
        assert _len != []                           # 我们假设 _len 是排好序的
        self.N = 1
        while self.N < len(_len):
            self.N *= 2
        self.c = [0] * (2 * self.N)                 # --- 覆盖
        self.s = [0] * (2 * self.N)                 # --- 总和
        self.w = [0] * (2 * self.N)                 # --- 长度
        for i in range(len(_len)):
            self.w[self.N + i] = _len[i]
        for p in range(self.N - 1, 0, -1):
            self.w[p] = self.w[2 * p] + self.w[2 * p + 1]

    def cover(self):
        return self.s[1]

    def change(self, i, k, delta):
        self._change(1, 0, self.N, i, k, delta)

    def _change(self, p, start, span, i, k, delta):
        if start + span <= i or k <= start:         # --- 分离
            return
        if i <= start and start + span <= k:        # --- 包含
            self.c[p] += delta
        else:
            self._change(2*p, start, span // 2, i, k, delta)
            self._change(2*p + 1, start + span // 2, span // 2, i, k, delta)
        if self.c[p] == 0:
            if p >= self.N:                         # --- 叶子节点
                self.s[p] = 0
            else:
                self.s[p] = self.s[2 * p] + self.s[2 * p + 1]
        else:
            self.s[p] = self.w[p]
```

算法的复杂度 $O(n\log n)$ 由横坐标和纵坐标的排序证明。

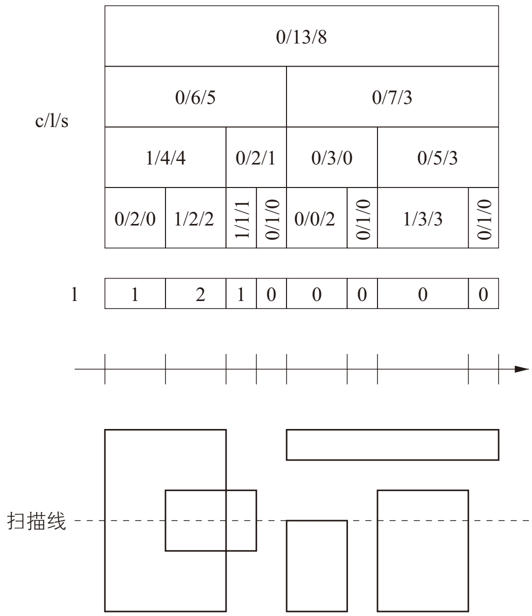


图 13.6　数据结构和扫描的图示

```
def union_rectangles(R):
    if R == []:
        return 0
    X = []
    Y = []
    for j in range(len(R)):
        (x1, y1, x2, y2) = R[j]
        assert x1 <= x2 and y1 <= y2
        X.append(x1)
        X.append(x2)
        Y.append((y1, +1, j))          # 生成事件
        Y.append((y2, -1, j))
    X.sort()
    Y.sort()
    X2i = {X[i]: i for i in range(len(X))}
    _len = [X[i + 1] - X[i] for i in range(len(X) - 1)]
    C = Cover_query(_len)
    area = 0
    last = 0
    for(y, delta, j) in Y:
        area += (y - last) * C.cover()
        last = y
        (x1, y1, x2, y2) = R[j]
        i = X2i[x1]
        k = X2i[x2]
        C.change(i, k, delta)
    return area
```

13.6 不相交长方形的合并

- **定义**

给定 n 个不相交的直线长方形，我们希望确定所有邻接的长方形对。

- **应用**

算法可以被用于计算并集的边长，实现方法是从长方形总边长中去除邻接长方形的接触边长。这一长度值必须带有一个系数 2，因为去除一段接触边长也就是去除每个长方形的一段边长。另一个应用是确定邻接长方形的连通分量（图 13.7）。

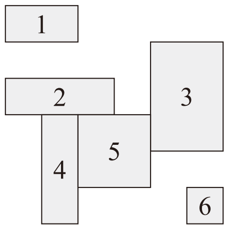


图 13.7 确定长方形 (2, 4)、(2, 5)、(3, 5) 和 (4, 5) 之间的邻接关系

- **复杂度为 $O(n\log n)$ 的扫描算法**

我们首先介绍如何利用长方形的左、右边来确定邻接长方形。上下邻接的情况也是一样的。首先用 n 个长方形中每个长方形的 4 个角点建立一个事件列表。每个事件是一个多元组 (x, y, i, c)，其中 (x, y) 是第 i 个长方形角点 c 的坐标，c 对于右下、右上、左下和左上的取值分别为 0、1、2、3。然后，所有事件按照字典序来处理，借此在一个列中从下往上、从左到右地扫描长方形的各个角点。当角点相同时，先处理在长方形顶边上的角点。

然后，我们只需维护一个已处理过底边角点，但还没有处理顶边角点的长方形列表[①]。由于长方形彼此不相交，这个列表中最多有两个长方形，一个位于左列，一个位于右列。一个底边角点将一个长方形加入这个列表，而一个顶边角点则将其移除。最终，当且仅当两个长方形同时出现在列表中时，它们才是邻接的。

- **变种**

如果在问题描述中，我们仅将共用一个角点而非一段邻边重合的长方形定义为邻接，那么我们需要改变处理事件的优先级，在处理两个相同的角点时优先处理底边角点，然后再处理顶边角点[②]。

① 因为是从下往上扫描。——译者注

② 因为是从下往上扫描，先处理底边角点，后处理顶边角点。于是，前面的长方形就不会因为顶边被处理而从列表中被移除。——译者注

第14章　计算

很多问题都能通过快速计算解决，比如素数的二项式系数问题。本章将一些高效解决如算数、表达式求值、线性系统求解等经典整数问题的实现方法进行了简单归类。

14.1　最大公约数

- **定义**

给定两个整数 a 和 b，我们寻找最大整数 p，使得 a 和 b 都可以表示为 p 的倍数，p 即为两个数的最大公约数。

最大公约数的计算可以使用递归方式快速实现。这里有一个记忆术：从第二次迭代开始，我们让第二个参数总是比第一个参数小，即 $a \bmod b < b$。

```
def pgcd(a, b)
    return a if b == 0 else pgcd(b, a%b)
```

14.2　贝祖等式

- **定义**

对于两个整数 a 和 b，我们希望确定两个整数 u 和 v，使得在 d 是 a 和 b 最大公约数的情况下有 $au + bv = d$。

这个计算基于一个简单结论。如果 $a = qb + r$、$au + bv = d$ 与 $(qb + r)\,u + bv = d$ 相关，对于 $bu' + rv' = d$ 有：

$$\begin{cases} u' = qu + v \\ v' = u \end{cases} \Leftrightarrow \begin{cases} u = v' \\ v = u' - qv' \end{cases}$$

这个计算可以在 $O(\log a+\log b)$ 个步骤后结束。其实，第一个参数每两步会减小一半。

- **变种**

有些问题更关心大数字的计算，因此需要返回一个大数除以一个大素数 p 的余数来确定结果是否成立。由于 p 是素数，我们可以把它除以一个非 p 倍数的整数 a：a 和 p 互质，所以其贝祖系数满足 $au + pu = 1$；因此 au 的值是 1 除以 p 求余，而 u 是 a 的倒数，所以 p 除以 a 也就是乘以 u。

```
def bezout(a, b):
    if b == 0:
        return (1, 0)
    else:
        u, v = bezout(b, a % b)
        return (v, u - (a // b) * v)

def inv(a, p):
    return bezout(a, p)[0] % p
```

14.3 二项式系数

当计算 $\binom{n}{k}$ 时，分别计算 $n(n-1),\cdots,(n-k+1)$ 和 $k!$ 是有风险的，因为可能会发生容量越界的情况。我们更倾向于使用以下结论：i 个连续整数的乘积一定包含一个能被 i 整除的元素。

```
def binom(n, k):
    prod = 1
    for i in range(k):
        prod = (prod * (n - i)) // (i + 1)
    return prod
```

在大多数问题中，计算二项式系数都需要除以一个素数 p。基于贝祖等式对系数的计算，代码如下，复杂度为 $O(k(\log k+\log p))$。

```
def binom_modulo(n, k, p):
    prod = 1
    for i in range(k):
        prod = (prod * (n - i) * inv(i + 1, p)) % p
    return prod
```

一个替代方案是使用动态规划来计算帕斯卡三角形。当 (n, k) 数对非常多时，这种方案在计算 $\binom{n}{k}$ 时很有意义。

14.4　快速求幂

- **定义**

给定 a 和 b，我们希望计算 a^b。重申一次，由于结果数值可能很大，通常会要求把算式的结果除以给定整数 q 并求余，但这不会改变问题的本质。

- **复杂度为 $O(\log b)$ 的算法**

简单解法会把 a 相乘 b-1 次。但我们可以利用关系 $a^{2k} \cdot a^{2k} = a^{2k+1}$ 来快速计算形式为 $a^1, a^2, a^4, a^8, \cdots$的 a 的幂。一个技巧就是把幂次 b 用二进制拆分，比如：

$$
\begin{aligned}
a^{13} &= a^{8+4+1} \\
&= a^8 \cdot a^4 \cdot a^1
\end{aligned}
$$

为了完成计算，只需生成 a 的 $O(\log_2 b)$ 次幂[①]。

```
def fast_exponentiation(a, b, q):
    assert a >= 0 and b >= 0 and q >= 1
    p = 0                               # 只用于记录
    p2 = 1                              # 2 ^ p
    ap2 = a % q                         # a ^ (2 ^ p)
    result = 1
    while b > 0:
        if p2 & b > 0:                  # b 由 a^(2^p) 拆分而来
            b -= p2
            result = (result * ap2) % q
        p += 1
        p2 *= 2
        ap2 = (ap2 * ap2) % q
    return result
```

- **变种**

这个技巧也可以用于矩阵乘法。设一个矩阵 A 和一个正整数 b，快速求幂算法能在 $O(\log b)$ 次矩阵乘法运算内计算 A^b。

14.5　素数

对于给定 n，我们寻找所有小于 n 的素数。“埃拉托斯特尼筛法”是实现目标的最简便方式。我们从一个所有小于 n 的整数列表开始：首先划掉 0 和 1；然后，对于每个 $p = 2, 3, 4, \cdots, n-1$，如果 p 没有被划掉，那么它就是素数；在这种情况下，我们把 p 的所有倍数都划掉。整个过程的复杂度分

① 这里只需计算二进制拆分后的最高次幂。因为在二进制拆分的过程中，计算最高次幂时已计算过所有比它小的幂，以便使用动态规划算法，利用已算好的结果。——译者注

析起来很繁琐，其值是 $O(n\log\log n)$。

- **实现细节**

下面提供的实现方法通过划掉 0、1 以及所有 2 的倍数来节省时间。在这种情况下，在对 p 进行迭代时的步长应当是 2，从而只测试所有奇数。

```
def eratosthene(n):
    P = [True] * n
    answ = [2]
    for i in range(3, n, 2):
        if P[i]:
            answ.append(i)
            for j in range(2 * i, n, i):
                P[j] = False
    return answ
```

14.6 计算算数表达式

- **定义**

给定一个遵循固定语法规则的表达式，我们希望建立语法树或者计算出该表达式的值（图 14.1）。

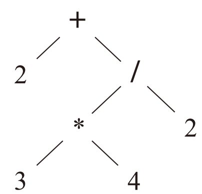


图 14.1 表达式 2 + (3 × 4) / 2 相关的树

- **方法**

一般来说，问题的解决方式是通过扫描器或分词器把包含表达式的字符串分割为词汇单元流。然后根据词汇单元，使用解析器来按照语法规则建立语法树。

但是，如果语法和算数表达式的语法类似，我们可以使用更容易实现的两个栈方法：一个栈保存值，另一个栈保存运算符。遇到一个数值，就将其原样存入保存值的栈。对于遇到的运算符 p，在把它加入运算符栈之前，需要执行以下操作：当运算符栈顶 q 的优先级至少和 p 相等时，我们把 q 出栈，最后两个值 a 和 b 也出栈，然后再把表达式 $a\ q\ b$（a 与 b 进行 q 运算的结果）的值加入数值栈。

使用同样的方式，运算符的求值被延期处理，直到优先级规则强制要求求值（图 14.2）。

读取	数值栈	运算符栈
2	2	∅
+	2	+
3	2,3	+
*	2,3	+,*
4	2,3,4	+,*
/	2,12	+
	2,12	+,/
2	2,12,2	+,/
;	2,6	+
	8	∅

图 14.2 处理表达式 $2+3\times4/2$ 的例子，使用“;”作为表达式终止符

- **电子表格的例子**

考虑一个电子表格，其每个格子可以保存一个值或一个算数表达式。算数表达式可以由常数值和格子的标识符组成，并由运算符号 - 、 + 、*、/ 和括号连接起来。

我们可以用一个整数、一个字符串或者由两个运算数和一个运算符组成的三元组来表示一个算数表达式。算数表达式的数值计算通过以下递归方法实现。其中 `cell` 是一个字典，将格子的名字与内容关联起来。

```
def arithm_expr_eval(cell, expr):
    if isinstance(expr, tuple):
    (left, op, right) = expr
    l = arithm_expr_eval(cell, left)
    r = arithm_expr_eval(cell, right)
    if op == '+':
        return l + r
    if op == '-':
        return l - r
    if op == '*':
        return l * r
    if op == '/':
        return l // r
    elif isinstance(expr, int):
        return expr
    else:
        cell[expr] = arithm_expr_eval(cell, cell[expr])
        return cell[expr]
```

语法树按照上述方法来建立。注意对括号的特殊处理：左括号总被加入运算符堆栈，而没有其他操作；右括号让运算符与相关左括号的顶点（即以它为根的子树）组成的表达式并出栈。为了在处理结束时彻底清空栈，我们在字符流尾部加入“;”作为表达式的结尾，并赋予它最低的优先级。

```
priority = {';': 0, '(': 1, ')': 2, '-': 3, '+': 3, '*': 4, '/': 4}

def arithm_expr_parse(line):
    vals = []
    ops = []
    for tok in line + [';']:
        if tok in priority:                    # tok 是一个运算符
            while tok != '(' and ops and priority[ops[-1]] >= priority[tok]:
                right = vals.pop()
                left = vals.pop()
                vals.append((left, ops.pop(), right))
            if tok == ')':
                ops.pop()                      # 这是与它相关的左括号
            else:
                ops.append(tok)
        elif tok.isdigit():                    # tok 是一个整数
            vals.append(int(tok))
        else:                                  # tok 是一个标识符
            vals.append(tok)
    return vals.pop()
```

- **陷阱**

在实现上述代码的过程中，大家会经常犯这样一个书写错误：

```
vals.append((vals.pop(), ops.pop(), vals.pop()))
```

鉴于 append 方法处理参数的顺序，这使得表达式左右两边的值相反[①]，导致我们不想要的效果。

14.7 线性方程组

- **定义**

线性方程组由 n 个变量和 m 个线性方程组成。正式来讲，给定一个维度为 $n \times m$ 的矩阵 A，和一个大小为 m 的列向量 b，目的是找到一个向量 x，使得 $Ax = b$。

- **应用：随机漫步**

假设一张连通图的每条弧上都标注了概率，离开弧的权重总和是 1。这样的图被称为**马尔科夫链**。随机漫步从一个顶点 u_0 开始，然后对于每个经过的顶点 u 都会有一条标注概率的弧 (u, v) 通过。我们想知道对于每个顶点 v，随机漫步到达 v 所需的时间 x_v。定义 $x_{u0} = 0$ 及 $x_v = \sum_u (x_u+1)p_{uv}$，其中 p_{uv} 是弧 (u, v) 上的概率；当不存在这条弧时，概率值为 0。

① 比如 a-b 处理成了 b-a。不同编程语言在处理方法或函数的参数列表时有不同的行为方式，有的从左往右处理，有的从右往左处理，写程序时要特别注意。——译者注

此外，还存在与随机漫步相关的另外一种应用。在 t 步以后，每个顶点都有一个出现漫步者的概率。在某些情况下，漫步会倾向于一种稳定的分布。计算这个分布归结为解决一个线性方程组问题，其中矩阵 A 主要编码了所有弧的概率，而我们寻找的时间 x 就是这个稳定分布的值。

- **应用：重物和弹簧系统**

假设一个系统中有一些通过弹簧连接的重球，弹簧自身的重量可以忽略。某些弹簧被挂在房顶，弹簧可以被拉伸或压缩（图 14.3）。给定球的位置和重量，我们想知道这个系统是稳定的，还是马上会运动。因此，目标就是找到每个弹簧两端受力的值，并尝试让重球所受的各种力能相互抵消，包括重力。这又重新回到了求解线性方程组问题。

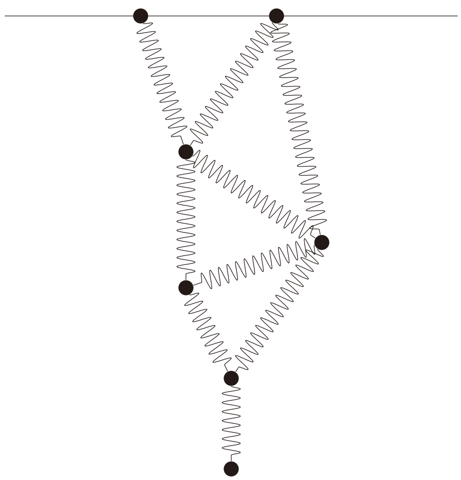


图 14.3　重物和弹簧系统

- **应用：地理交叉问题**

在地理学中，线和超平面是以线性方程定义的。确定它们在何处交叉，也是求解线性方程组的问题。

- **复杂度为 $O(n^2m)$ 的算法**

如果 A 是单位矩阵[①]，系统的解答是 b。我们要诊断 A 来获取与理想解答最接近的解。

为此，我们需要应用一些能保留系统解的变换，比如交换变量的顺序、交换两个多项式、把一个多项式的两边乘以一个常数、把两个多项式相加。

为了简化数据操作且不修改 A 和 b 参数，我们把这个多项式系统储存到矩阵 S 里。S 由 A 的一个副本组成，我们在副本中添加一个列，其中包含 b 和包含变量下标的额外一行。因此，当各个列在 S 中彼此交换时，我们能发现每一列与哪个变量相关。

不变量如下：在 k 次迭代后，S 的前 k 列都会变成 0，除了值为 1 的对角线上的元素（图 14.4）。

① 单位矩阵是一个方阵，从它的左上角到右下角的对角线上的元素均为 1。除此以外全部元素为 0。——译者注

为了得到这个不变量，我们把 S 的第 k 行，也就是第 k 个多项式除以 S[k, k]（当它非空时），就此给下标 (k, k) 值添加 1。然后，把所有 $i \neq k$ 的多项式中第 k 个多项式减掉，并乘以因子 S[i, k]，使得当 $i \neq k$ 时，给下标 (i, k) 值添加 0，如同不变量要求的那样。

如果 S[k, k] 值为空，该怎么办？在这些操作开始之初，我们在 S[k, k] 和 S[m–1, n–1] 围成的长方形内寻找一个绝对值最大的元素 S[i, j]，然后交换 k 列和 j 列，以及 k 行和 i 行。这些操作会保留系统的解。

如果这个长方形只包含 0，又该怎么办？在这种情况下，对角化操作会终止，并用如下方式提取最终解。

如果对角化在 k 次迭代后过早地结束，而且 S 中从 k 到 m–1 行都是 0，那么需要检查这些行的最后一列。如果存在一个输入值是非空值 v，那么意味着存在悖论 $0 = v$，因此我们可以断定这个问题没有解；否则，这个系统至少有一个解。假设已经执行了 k 次迭代对角化，我们有 $k \leqslant \min\{n, m\}$。如果 $k < n$，那么系统有多个解。为了选择一个解，我们把 S 中从 k 到 n–1 列中的相应变量设置为 0，然后就可以在最后一列得到其他变量的其他值，并由 S 中值为 1 的对角线来限定。最终如果 $k = n$，那么解是唯一的。

$$A = \begin{pmatrix} 1 & 0 & 0 & \cdot & \cdot & \cdot \\ 0 & 1 & 0 & \cdot & \cdot & \cdot \\ 0 & 0 & 1 & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & \cdot & \cdot & \cdot \end{pmatrix} b = \begin{pmatrix} \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \end{pmatrix}$$

图 14.4　当 $k = 3$ 时不变量的结构

- **实现细节**

由于使用浮点数计算，计算中会存在精度问题，因而我们在测试等于 0 的时候需要加入阈值。在切分第 k 行到第 i 行的时候，需要把系数 S[i][k] 赋值给一个变量 `fact`，因为这个操作会改变 S[i][k] 的值。为了精确地计算结果，我们可以使用分数，而不再是浮点数。在这种情况下，在每次迭代时化简分数是非常重要的，否则解的分子和分母会包含指数级的数字数量。在不规范化的情况下，高斯 – 若尔当消元法不是多项式时间复杂度的，但所幸 Python 的类库 `Fraction` 在每次操作中化简了分数。

- **变种**

当矩阵是稀疏矩阵时，也就是当矩阵每行、每列中的非零元素非常少时[①]，我们可以把计算时间从 $O(n^2m)$ 减少到 $O(n)$。

---

① 换句话说，当 0 元素数目远远多于非 0 元素的数目，并且非 0 元素分布没有规律时，矩阵被称为稀疏矩阵。——译者注

```
def is_zero(x):                              # 阈值
    return -1e-6 < x and x < 1e-6            # 如果使用分数计算，替换为 x == 0

GJ_ZERO_SOLUTIONS = 0                        # 返回值
GJ_UNE_SOLUTION = 1
GJ_PLUSIEURS_SOLUTIONS = 2

def gauss_jordan(A, x, b):
    n = len(x)
    m = len(b)
    assert len(A) == m and len(A[0]) == n
    S = []                                   # 把系统放入一个唯一的矩阵 S
    for i in range(m):
        S.append(A[i][:] + [b[i]])
    S.append(list(range(n)))                 # x 中的下标
    k = diagonalize(S, n, m)
    if k < m:
        for i in range(k, m):
            if not is_zero(S[i][n]):
               return GJ_ZERO_SOLUTIONS
    for j in range(k):
        x[S[m][j]] = S[j][n]
    if k < n:
        for j in range(k, n):
            x[S[m][j]] = 0
        return GJ_PLUSIEURS_SOLUTIONS
    return GJ_UNE_SOLUTION

def diagonalize(S, n, m):
    for k in range(min(n, m)):
        val, i, j = max((abs(S[i][j]), i, j)
                         for i in range(k, m) for j in range(k, n))
        if is_zero(val):
            return k
        S[i], S[k] = S[k], S[i]              # 交换 k 行 j 行
        for r in range(m + 1):               # 交换 k 列 j 列
            S[r][j], S[r][k] = S[r][k], S[r][j]
        pivot = float(S[k][k])               # 如果使用分数计算，不需要 float
        for j in range(k, n + 1):
            S[k][j] /= pivot                 # 把 k 行除以 pivot
        for i in range(m):                   # 去掉 i 行到 k 行
            if i != k:
               fact = S[i][k]
               for j in range(k, n + 1):
                   S[i][j] -= fact * S[k][j]
    return min(n, m)
```

14.8 矩阵序列相乘

- **定义**

设有 n 个矩阵 $M_1,\cdots,M_n$，其中第 i 个矩阵有 r_i 行和 c_i 列，且对所有 $1\leqslant i<n$，有 $c_i=r_{i+1}$。我们希望用最少次数的操作来计算 $M_1M_2\cdots M_n$ 的乘积。通过结合律，存在多种添加括号的方式。通过矩阵乘积的结合律得到以下等式，但这些计算的复杂度可能会不同。

$$(((M_1M_2)M_3)M_4=M_1(M_2(M_3(M_4)))=(M_1M_2)(M_3M_4),$$

为了把两个矩阵 M_i 和 M_{i+1} 相乘，我们使用执行 $r_ic_ic_{i+1}$ 次数字乘法的标准算法。目的是找到放置括号的方法，从而用最小的成本执行乘法（图 14.5）。

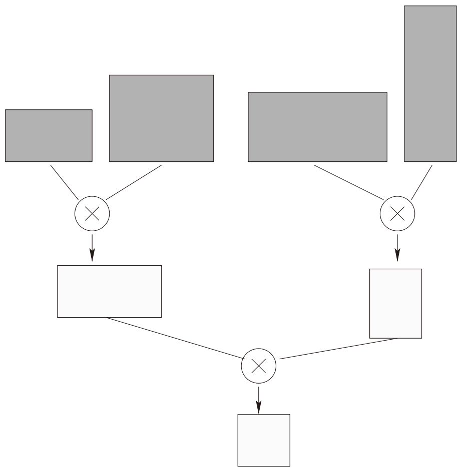


图 14.5 用哪个括号做矩阵序列乘法，可以让操作次数最少？

- **复杂度为 $O(n^2)$ 的算法**

循环公式很简单[①]。对于某个 $1\leqslant k<n$，最后一个乘法把 $M_1,\cdots,M_k$ 的结果与 $M_{k+1},\cdots,M_n$ 的结果相乘。其中 opt(i,j) 是计算 $M_i,\cdots,M_j$ 的最小成本。因此我们有 $s(i,i)=0$，且当 $i<j$ 时

$$\text{opt}(i,j)=\min_{i\leqslant k<j}(\text{opt}(i,k)+\text{opt}(k+1,j)+r_ic_kc_j. \qquad (14.1)$$

如果既想计算最优顺序的成本，又想计算最优顺序本身，那么必须在 opt 矩阵中增加下标 k 来实现最小化（14.1）。这正是以下实现所做的。函数 `opt_mult(M, opt, i, j)` 根据 opt 中存储的信息以最优方式按顺序计算 $M_i,\cdots,M_j$。

注意下标 i 和 j 的处理顺序。考虑 $j-i$ 的升序，可以保证的是确定公式 14.1 中最小值所需的数对 (i,k) 和 $(k+1,j)$ 值已经计算完成。

① 复杂度仅和括号位置的计算有关，与矩阵乘法本身无关。

```
def matrix_mult_opt_order(M):
    n = len(M)
    r = [len(Mi) for Mi in M]
    c = [len(Mi[0]) for Mi in M]
    opt = [[0 for j in range(n)] for i in range(n)]
    arg = [[None for j in range(n)] for i in range(n)]
    for j-i in range(1, n):                    # 从 j-i 开始降序对 i 循环
        for i in range(n - j_i):
            j = i + j_i
            opt[i][j] = float('inf')
            for k in range(i, j):
                alt = opt[i][k] + opt[k + 1][j] + r[i] * c[k] * c[j]
                if opt[i][j] > alt:
                    opt[i][j] = alt
                    arg[i][j] = k
    return opt, arg

def matrix_chain_mult(M):
    opt, arg = matrix_mult_opt_order(M)
    return _apply_order(M, arg, 0, len(M)-1)

def _apply_order(M, arg, i, j):
    # --- 包含矩阵 M[i] 到 M[j] 的乘法
    if i == j:
        return M[i]
    else:
        k = arg[i][j]                          # 根据放置括号的结果进行
        A = _apply_order(M, arg, i, k)
        B = _apply_order(M, arg, k + 1, j)
        row_A = range(len(A))
        row_B = range(len(B))
        col_B = range(len(B[0]))
        return[[sum(A[a][b] * B[b][c] for b in row_B)
                for c in col_B] for a in row_A]
```

有一个更好的复杂度为 $O(n\log n)$ 的算法，这里就不多做介绍了（见参考文献 [15]）。

第 15 章　穷举

对于有些组合问题，没有能保证在多项式时间内解决问题的已知数据结构。这时，需要一个遍历所有潜在答案空间的穷举法。这里的“组合”指的是简单数据结构组成较复杂数据结构，例如子树构建树，用瓦片来铺路，等等。所以，穷举法意味着遍历所有可能构建的隐性树，借此找到一个解。但是，树的节点表示部分结构，如果部分结构不能形成一个完整的解，比如不满足某个限制条件，那么遍历会返回上一级节点，尝试另外一个分支。因此，这种方法叫“回溯法”（backtracking）。我们会用一个简单例子来介绍。

15.1　激光路径

- **定义**

假设一个长方形网格，它被一个在第一行顶边左侧和右侧有开口的边界包围。网格的某些格子中有双面镜子，可以用对角线和反对角线方式布置（图 15.1）。我们的目的是把镜子按照某种方式布置，使得激光光束从左侧开口进入，从右侧开口离开。光束在网格中横向或纵向穿过，当它碰到镜子时，会根据镜子的方向向左或向右转 90°。如果激光碰到网格的边界，就会被吸收。

- **算法**

这个问题没有任何在多项式时间内解决的已知算法。我们建议使用穷举法来实现。对每个镜子存储一个状态，共有三种可能类型的状态：两个方向，以及“没有方向”。起初，所有镜子都没有方向；然后，模拟激光光束从左侧开口进入，并在网格间穿过。

— 当光束碰到一个有方向的镜子，光束会根据镜子的方向被反射。

— 当光束碰到一个没有方向的镜子，我们要执行两个递归调用，即对镜子每个可能方向分别执行一个调用。如果其中一个调用找到了一个解，那么它被返回。如果任何一个调用都没

有找到解，这面镜子就被重新放回没有方向的状态，一个代表失败的代码被返回。

— 当光束碰到网格边界，递归过程结束并返回一个代表失败的代码。穷举过程返回上一层，也就是“回溯”。

— 最终，光束到达右侧开口，返回解。

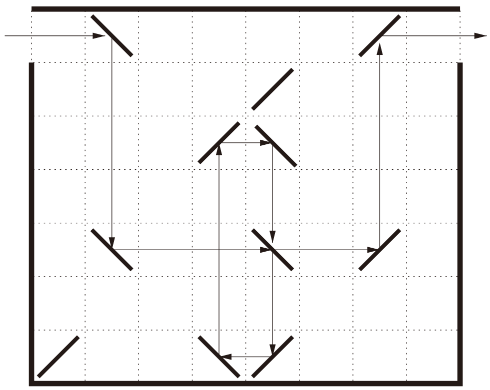


图 15.1　能让激光通过网格的一种镜子放置方式

● **实现细节**

在输入中给定了初始位置的 n 个镜子被从 0 到 n-1 编号。两个假想镜子被编号为 n 和 n+1，放在两个开口处。程序生成一个数组 L，其中有镜子的坐标和下标。

光束的 4 个可能方向以 0 至 3 这 4 个整数编码，镜子的两个方向编码为 0 和 1。一个维度为 4 × 2 的数组 reflex 表示光束以一个给定方向抵达一个给定反射方向的镜子时发生的方向变化。

在预先计算中，在一行或一列中的一系列连续镜子通过数组 succ 相关联。对于一个镜子 i 和一个方向 d，当光束按 d 方向离开了镜子 i，输入 succ[i][d] 表示光束遇到的下一个镜子的下标。当光束被反射到边界时，这个输入值是 `None`。

为了填充数组 succ，首先要一行行、一列列地遍历数组 L。注意，函数 `tri` 使用字典序排序来翻转行下标和列下标。

变量 last_i、last_r 和 last_c 保存了在遍历中遇到的最后一个镜子的信息。如果这面镜子与当前镜子在同一行（按行遍历时），那么需要在 succ 中把两者编号设置为关联。

```
# 方向
UP = 0
LEFT = 1
DOWN = 2
RIGHT = 3
# 镜子的方向  None :? 0:/ 1:\

# 光束来的方向 UP             LEFT         DOWN             RIGHT
reflex = [[RIGHT, LEFT], [DOWN, UP], [LEFT, RIGHT], [UP, DOWN]]     # 与上一行光束来的
方向对应的反射方向
```

```
def laser_mirrors(rows, cols, mir):
    # construire les structures
    n = len(mir)
    orien = [None] * (n + 2)
    orien[n] = 0                              # 开口格子里镜子的方向是随机的
    orien[n + 1] = 0
    succ = [[None for direc in range(4)] for i in range(n + 2)]
    L = [(mir[i][0], mir[i][1], i) for i in range(n)]
    L.append((0, -1, n))                      # 进入
    L.append((0, cols, n + 1))                # 离开
    last_r = None
    for(r, c, i) in sorted(L):                # 按行扫描
        if last_r == r:
            succ[i][LEFT] = last_i
            succ[last_i][RIGHT] = i
        last_r, last_i = r, i
    last_c = None
    for(r, c, i) in sorted(L, key= lambda tup_rci : (tup_rci[1], tup_rci[0])):
        if last_c == c:                       # 按列扫描
            succ[i][UP] = last_i
            succ[last_i][DOWN] = i
        last_c, last_i = c, i
    if solve(succ, orien, n, RIGHT):          # 遍历
        return orien[:n]
    else:
        return None
```

遍历是通过递归调用实现的。对于此等难度的问题，实例一般都比较小，使用递归调用时不存在堆栈溢出的问题。注意，在对镜子 j 的两个可能方向所对应的两个子树执行无效遍历以后，程序会重置变量内容，即改为无方向状态。

```
def solve(succ, orien, i, direc):
    assert orien[i] != None
    j = succ[i][direc]
    if j is None:                     # 基本情况
        return False
    if j == len(orien) - 1:
        return True
    if orien[j] is None:              # 测试镜子的 2 个方向
        for x in[0, 1]:
            orien[j] = x
            if solve(succ, orien, j, reflex[direc][x]):
                return True
        orien[j] = None
        return False
    else:
        return solve(succ, orien, j, reflex[direc][orien[j]])
```

15.2　精确覆盖

舞蹈链算法是穷举算法中的劳斯莱斯，它能解决通用的精确覆盖问题。很多问题都能化简为精确覆盖问题，因此，掌握这种算法在竞赛中无疑是一个实实在在的优势。

- **定义**

精确覆盖问题由一个点的集合 U（称作空间）以及一个 U 的子集 S ⊆ 2^U 组成（图 15.2）。当 *x* ⊆ A 时，我们称一个集合 A ⊆ U 覆盖了 *x*∈U。目的是找到 S 的一个选择集，即一个集合 S* ⊆ S，使它精确覆盖整个空间中的每个元素一次。

在输入中，我们收到一个二进制矩阵 M，矩阵的列代表了整个空间中的所有元素，行代表 S 的所有集合。在 *x*∈A 时，矩阵中的元素 <*x*，A> 值为 1。在输出中，需要生成一个行的集合 S*，使得被限制在 S 里的矩阵在每列精确地包含一个 1。

- **应用**

数独游戏可以被视为一个精确覆盖问题（见 15.3 节）。铺路问题也是如此，即在一个地砖集合中，如何覆盖一个维度为 $m \times n$ 的网格且没有交叉。每块地砖必须被精确地使用一次，每个待铺的格子必须被一块地砖覆盖。因此，格子和地砖形成了一个空间元素，而地砖的铺设方式形成了集合。

- **舞蹈链算法**

算法实现的就是上述穷举遍历。其特色是在实现中选择数据结构。

首先，算法选择一个元素 *e*，该元素在 S 的最小集合中，也就是受限制最多的集合。这个选择很有可能会生成一些小查找树。由于解必须覆盖 *e*，因而一定包含且仅包含一个集合 A，满足 A∈S 且 *e*∈A。因此，解的搜索空间被子集的选择拆分。对于每个满足 *e*∈A 的集合 A，我们为子问题搜索一个解 S，在找到这个解的情况下，解 S* ∪ {A} 作为初始问题的一个解被返回。

从 U 中去掉 A 的元素后，可以从 <U, S> 建立待解决的子问题，因为每个元素 *f*∈A 已被 A 覆盖，而且不能覆盖第二次。另外，A 与 B 的所有交叉元素已从 S 中去掉，否则 A 中的元素会被覆盖多次（图 15.2）。

为了形式化包含矩阵 M，重建后的算法基本结构如下。如果矩阵 M 为空，那么需要返回空集合，它是一个解；否则，寻找一个 M 中可能包含至少一个 1 的列 *c*。在所有可能覆盖 *c* 的列 *r* 上，也就是 $M_{rc} = 1$ 的列上进行循环。对每个行 *r* 执行下列操作：从 M 中去掉行 *r* 和所有被 *r*（$M_{rc'} = 1$）覆盖的列 *c'*。如果所得矩阵有一个解 S，那么返回 S ∪ {r}；否则恢复 M。

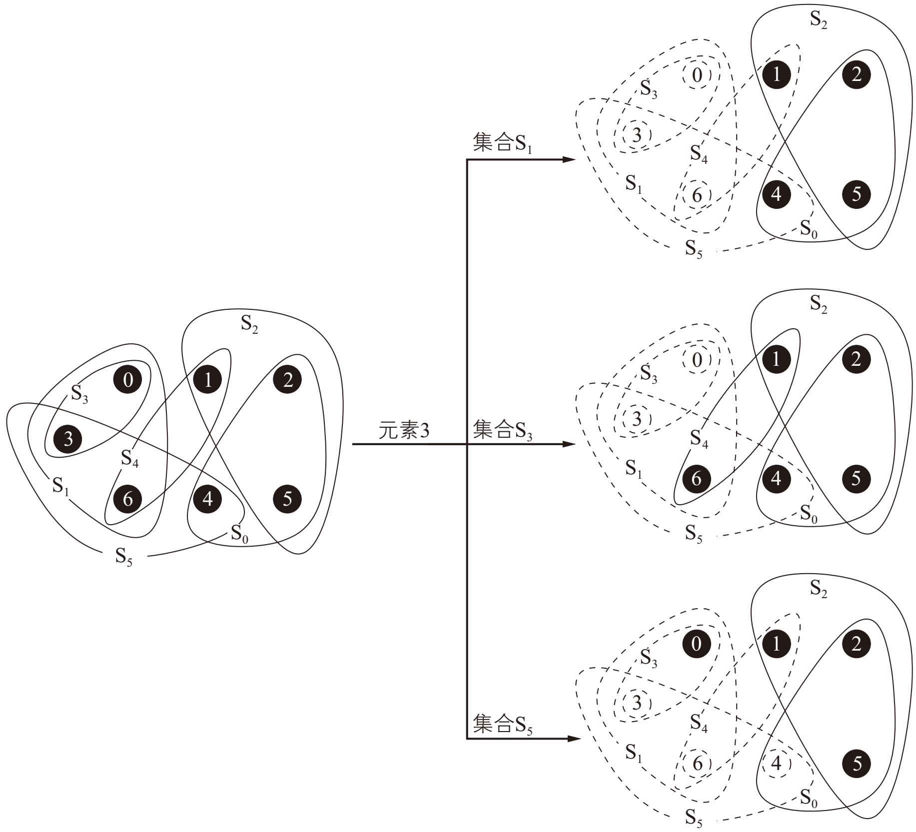


图 15.2　选择元素 e = 3 带来的三个子问题。注意，第三个子问题没有解，因为元素 0 无法被覆盖

- **链式简约表示方法**

为了简约表示稀疏矩阵 M，我们只保存 M 中包含一个 1 的格子，并将它们通过横向和纵向的两个链关联起来（图 15.3）。这样一来，我们可以轻松通过横向链来遍历所有满足 $M_{rc} = 1$ 的列 r。每个格子有四个字段 L、R、U、D 来编码双重链。

每个列还有一个头部格子，它是纵向链的一部分，让我们可以访问列。在建立结构时，头部格子被存储在一个用列编号索引的数组 col 中。然后，我们得到一个特殊格子 h，它是纵向链的一部分，用来存储头部格子以便访问列。这个格子不使用字段 U 和 D。

每个格子有两个额外字段 S 和 C，其作用和格子的类型有关。对于矩阵的格子，S 保存行的编号，C 保存列的头部格子。对于列的头部格子，S 保存列中 1 的数量，字段 C 被忽略。格子 h 会忽略这两个字段。

```python
class Cell:
    def __init__(self, horiz, verti, S, C):
        self.S = S
        self.C = C
        if horiz:
            self.L = horiz.L
            self.R = horiz
            self.L.R = self
            self.R.L = self
        else:
            self.L = self
            self.R = self
        if verti:
            self.U = verti.U
            self.D = verti
            self.U.D = self
            self.D.U = self
        else:
            self.U = self
            self.D = self

    def hide_verti(self):
        self.U.D = self.D
        self.D.U = self.U

    def unhide_verti(self):
        self.D.U = self
        self.U.D = self

    def hide_horiz(self):
        self.L.R = self.R
        self.R.L = self.L

    def unhide_horiz(self):
        self.R.L = self
        self.L.R = self
```

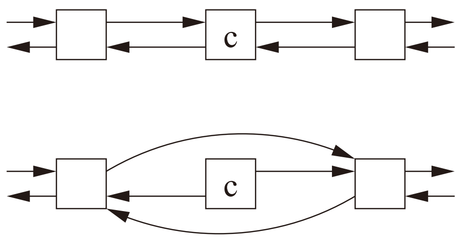
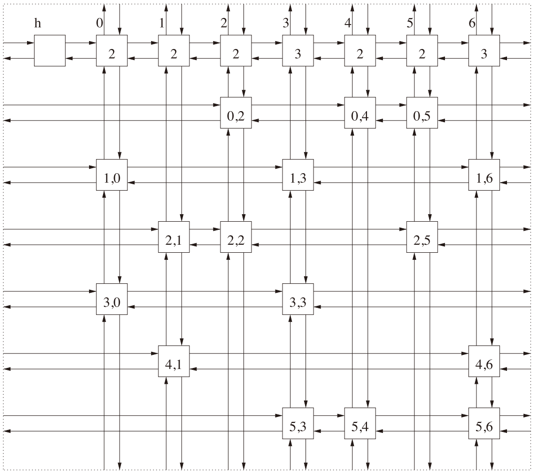


图 15.3　一个二进制矩阵 M，上方是它的编码，下方是覆盖第 0 列的结果。链都是循环的，链从图的一边离开，再从相对的另一边重新进来

- **链**

舞蹈链算法的思路源于一松宏和野下浩平在 1979 年发现并由高德纳在 2000 年描述的研究结果（见参考文献 [18]）：为了从一个双向链表中提取出一个元素 c，只需改变其相邻指针（图 15.4）；为了把元素重新加入链表，只需按相反顺序执行反向操作。



$$\begin{pmatrix} \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

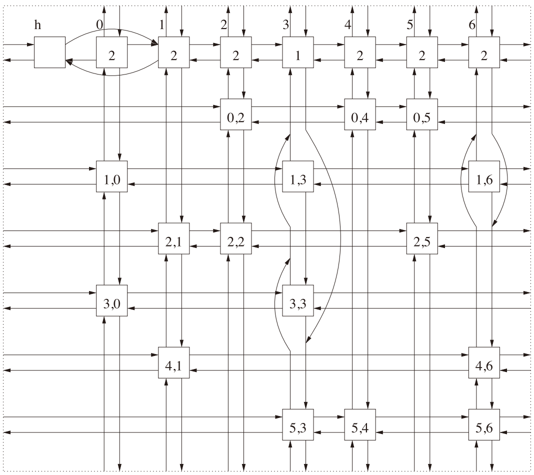


图 15.4 `hide` 操作从一个双向链中去掉一个元素 c。不删除 c 的指针，很容易就能把该元素重新插入其初始位置

我们利用这一结果在矩阵中去掉或添加一个列。覆盖一个列 c，就是把它从所有列的头部元素所在横行中移除；同时，对于满足 $M_{rc} = 1$ 的行 r，从纵向链中去掉在 M（$M_{rc'}=1$）中所有与位置 (r, c') 相关的格子。注意随时维护格子 c' 头部的计数器 S，即减少它的值。

```
def cover(c):                            # c = 要隐藏的列的头部元素
    assert c.C is None                   # 必须是一个头部格子
    c.hide_horiz()
    i = c.D
    while i != c:
        j = i.R
        while j != i:
            j.hide_verti()
            j.C.S -= 1                   # 这个列中减少一个元素
            j = j.R
        i = i.D

def uncover(c):
    assert c.C is None
    i = c.U
    while i != c:
        j = i.L
        while j != i:
            j.C.S += 1                   # 这个列中增加一个元素
            j.unhide_verti()
            j = j.L
        i = i.U
    c.unhide_horiz()
```

- **搜索**

在搜索过程中，我们仅遍历所有列来找到令元素最少的列，即最小的计数器 S。我们用列的优先级队列来加速操作，但列的覆盖操作成本会更高。在函数返回 `true` 时，以下函数把解写入数组 sol。

```
def dancing_links(size_universe, sets):
    header = Cell(None, None, 0, None)       # 建立格子的结构
    col = []
    for j in range(size_universe):
        col.append(Cell(header, None, 0, None))
    for i in range(len(sets)):
        row = None
        for j in sets[i]:
            col[j].S += 1                    # 这一列中增加一个元素
            row = Cell(row, col[j], i, col[j])
    sol = []
    if solve(header, sol):
        return sol
    else:
        return None
```

```
def solve(header, sol):
    if header.R == header:                    # 空的输入值，找到答案
        return True
    c = None                                  # 搜索最小覆盖的列
    j = header.R
    while j != header:
        if c is None or j.S < c.S:
            c = j
        j = j.R
    cover(c)                                  # 覆盖这一列
    r = c.D                                   # 尝试这一列
    while r != c:
        sol.append(r.S)
        j = r.R                               # 在元素 r 中覆盖元素
        while j != r:
            cover(j.C)
            j = j.R
        if solve(header, sol):
            return True
        j = r.L                               # 还原
        while j != r:
            uncover(j.C)
            j = j.L
        sol.pop()
        r = r.D
    uncover(c)
    return False
```

15.3 数独

很多贪婪算法都能高效地解决经典数独问题（图 15.5）。但对于 16 × 16 的网格数独问题来说，舞蹈链算法更合适。

7	2							
	5				9			
				3	8			
			4			5		
		3				8		
		2			3			
			2	5				
			6				3	
							1	9

图 15.5　一道网格数独题。目的是把空格子填满，满足每行、每列和每个 3 × 3 方块都包含从 1 到 9 的所有整数

- **建模**

如何把一道数独题建模为一个精确覆盖问题？有 4 个限制条件：每个格子必须有一个值；在数独网格的每一行、每一列和每一个方块，每个值只能精确地出现一次。因此，在空间中有 4 种元素：行列对、行值对、列值对和块值对。这些元素组成了精确覆盖问题实例 <U, S> 的空间 U。

现在，S 的集合将构成赋值法，也就是“行 – 列 – 值”三元组。每个赋值精确覆盖空间汇总的 4 个元素。

这一描述用简洁、便利的方式抽象表达了行、列、块和值，让问题更容易处理。

如何在实例中给数独网格中有固定值的格子编码呢？我们的方法是在空间中添加一个新元素 e，并在 S 中添加一个新集合 A——它是唯一一个包含 e 的集合。因此，所有结果必须由 A 组成。接下来只需在 A 中填入空间中被初始赋值覆盖了的所有元素。

- **编码**

精确覆盖实例 <U, S> 的集合与赋值相关，而舞蹈链算法的实现以被选中元素的下标数组形式来返回结果。因此，为了找到与下标相关的赋值，必须明确一种编码方法。将 v 值填入行 r、列 c 的格子，我们将这一赋值编码为 $81r + 9c + v$（对于 16 × 16 的数独网格，把参数替换为 256 和 16）。

同样，空间中元素也被编码成整数，比如行列对 (r, c) 被编码为 $9r + c$，列值对 (r, v) 被编码为 $81 + 9r + v$，以此类推。

```
N = 3          # 全局常量
N2 = N * N
N4 = N2 * N2

# 集合
def assignation(r, c, v): return r * N4 + c * N2 + v

def row(a): return a // N4
def col(a): return(a // N2) % N2
def val(a): return a % N2
def blk(a): return(row(a) // N) * N + col(a) // N

# 待覆盖元素
def rc(a): return row(a) * N2 + col(a)
def rv(a): return row(a) * N2 + val(a) + N4
def cv(a): return col(a) * N2 + val(a) + 2 * N4
def bv(a): return blk(a) * N2 + val(a) + 3 * N4


def sudoku(G):
    global N, N2, N4
    if len(G) == 16:                 # 对于 16 × 16 的数独问题
        N, N2, N4 = 4, 16, 256
    e = 4 * N4
    univers = e + 1
    S = [[rc(a), rv(a), cv(a), bv(a)] for a in range(N4 * N2)]
```

```
    A = [e]
    for r in range(N2):
        for c in range(N2):
            if G[r][c] != 0:
                a = assignation(r, c, G[r][c] - 1)
                A += S[a]
    sol = dancing_links(univers, S + [A])
    if sol:
        for a in sol:
            if a < len(S):
                G[row(a)][col(a)] = val(a) + 1
        return True
    else:
        return False
```

15.4 排列枚举

- **应用**

一些缺乏结构的问题需要用穷举法解决，在所有潜在解范围内逐个测试每个元素。因此，有时需要遍历一个给定数组的所有排列。

- **例子：单词相加**

考虑下面格式的问题：

```
      S E N D
+     M O R E
=   M O N E Y
```

给每个字符赋予一个唯一的数字，使得每个单词成为一个开头不为 0 的数字，并令加法等式成立。用穷举法解决问题时，只需建立一个由问题中字母组成的数组 tab = “@@DEMNORSY”，并用足够多的 @ 将数组补全到 10 个字符。现在，把每个字母和其在数组中的位置关联，令数组排列和字母的赋值之间有了相关性。

其中有意义的是枚举一个列表的所有排列，这正是本章的主题。

- **定义**

给定一个有 n 个元素的数组 t，我们希望确定 t 之后的一个字典序排列，或者确定 t 已经是最大的。

- **关键测试**

为了把 t 排列成其身后的字典序数组，我们想保留最长的前缀，而且只在后缀中交换元素。

● 线性时间复杂度的算法

算法基于三个步骤。第一，需要找到最大下标 p（称为“轴”，pivot），使得 $t[p]<t[p+1]$。思路是由于从 p+1 开始的 t 的后缀是一个非增长序列，因此该后缀已是字典序中最大的；所以，如果不存在这样一个轴，算法即可宣告 t 是最大字符，并结束。

显然，此时 $t[p]$ 应当增长，然而是以最小化的方式增长。因此，我们在后缀中寻找一个下标 s 使得 $t[s]$ 最小，且有 $t[s]<t[p]$。因为 p+1 是候选者，所以这样一个下标总存在。在把 $t[s]$ 和 $t[p]$ 交换以后，我们得到一个字典序大于初始数组的数组。最终，从 p+1 开始把 t 的后缀按升序排列，借此，我们获得在前缀 $t[1\cdots p]$ 中最小的排列（图 15.6）。

初始数组	0	2	1	6	5	2	1
选择轴	0	2	[1]	6	5	2	1
交换	0	2	[2]	6	5	[1]	1
反转	0	2	2	[1	1	5	6]
最终数组	0	2	2	1	1	5	6

图 15.6　计算后续的排列

将后缀按照升序排列又变回将其元素反转的操作，因为最初元素是降序排列的。

```python
def next_permutation(tab):
    n = len(tab)
    pivot = None                                # 找到轴
    for i in range(n - 1):
        if tab[i] < tab[i + 1]:
            pivot = i
    if pivot is None:                           # 数组已是最大
        return False
    for i in range(pivot + 1, n):               # 确定待交换元素
        if tab[i] > tab[pivot]:
            swap = i
    tab[swap], tab[pivot] = tab[pivot], tab[swap]
    i = pivot + 1
    j = n - 1                                   # 把后缀反转
    while i < j:
        tab[i], tab[j] = tab[j], tab[i]
        i += 1
        j -= 1
    return True
```

因此，单词相加问题的解可以用如下方式编码：

```python
def convert(word, ass):
    retval = 0
    for x in word:
        retval = 10 * retval + ass[x]
    return retval

def solve_word_addition(S):                     # 返回解的数字
```

```
    n = len(S)
    letters = sorted(list(set(''.join(S))))
    not_zero = ''                                    # 它不能是 0
    for word in S:
        not_zero += word[0]
    tab = ['@'] * (10-len(letters)) + letters        # 最大字典序排列
    count = 0
    while True:
        ass = {tab[i]: i for i in range(10)}         # 相关数组
        if tab[0] not in not_zero:
            sum = - convert(S[n-1], ass)             # 相加
            for word in S[:n-1]:
                sum += convert(word, ass)
            if sum == 0:                             # 计算是否正确?
                count += 1
        if not next_permutation(tab):
            break
    return count
```

- **变种：组合和排列的枚举**

在 n 个元素中枚举 k 种元素组合，即 $\{1,\cdots,n\}$ 中有 k 个元素的部分，技巧是在二进制掩码“取 k，不取 $n-k$”的排列上进行迭代。这个掩码也就是由 $n-k$ 个元素 0 及其后续的 k 个元素 1 组成的数组，因此它能让我们选择保存在子集中的元素。

为了枚举 n 个元素的 k 种排列方式，只需枚举 n 个元素的 k 种元素组合，这就回到了使用两个嵌套迭代 `next_permutation` 的解决方法。这给我们提供了另一种解决单词相加问题的解法：选择 10 个字母中 k 个字母的排列方式，其中 k 是不同字母的数量。

我们还将介绍一种技术，其实它已在 1.6.6 节中提到过。这种技术能更有技巧性地遍历一个有 n 个元素的集合的各个部分，以此来解决一大类动态规划问题。

15.5 正确计算

这一问题来自法国电视节目《数字和字母》中一个著名的游戏。

输入：n+1 个整数 $x_0,\cdots,x_{n-1}$，b 是一个比 n 小的数，设 $n \leqslant 20$。

输出：一个算数表达式最多使用每个整数一次，采用任意次加减乘除运算符，令计算结果尽可能接近 b。减法只允许在结果为正整数时使用，除法只允许在结果能被整除时使用。

- **复杂度为 $O(3^n)$ 的算法**

算法通过穷举法和动态规划法实现。在一个字典 E 中，我们把 $S \subseteq \{1,\cdots,n-1\}$ 与最多使用一次输入 x_i（$i \in S$）计算所得的结果关联[①]。具体来讲，$E[S]$ 成了把每个可得结果值与表达式相

① 除了 $S=\varnothing$，它被忽略了。

关联的字典。

比如，对于 $x = (3, 4, 1, 8)$ 且 S = {0, 1}，字典 $E(S)$ 包含了键 x 和值 e 的数值对，其中 e 是由输入 $x_0 = 3$ 和 $x_1 = 4$ 组成的表达式，而 x 是 e 的值。因此 $E[s]$ 包含了键值对 $1 \rightarrow 4\text{-}3$、$3 \rightarrow 3$、$4 \rightarrow 4$，$7 \rightarrow 3\text{+}4$ 和 $12 \rightarrow 3 \times 4$。

为了计算 $E[S]$，我们在 S 的两个非空集 L 和 R 分段上进行循环。对于 $E[L]$ 中通过一个表达式 e_L 即可得到的每个值 v_L，以及 $E[R]$ 中通过一个表达式 e_R 即可得到的每个值 v_R，我们都可以重建新值并存储于 $E[S]$ 中。尤其，v_L+v_R 可以通过表达式 e_L+e_R 计算得到。

算法的复杂度可以用如下方式评估。对于每个基数 k，考虑 $\binom{n}{k}$ 个满足 $|S|=k$ 的集合 S。对于每个集合 S，其所有子集 L 要被考虑，后者数量是 2^k。固定的 S 和 L 所需工作量是常数，因此算法复杂度是 $\sum_{k=1}^{n}\binom{n}{k}2^k = 0(3^n)$ 。

在处理 S 的子集时要特别注意，必须遵守基数的升序排序。这样，我们可以保证所有集合 $E[L]$ 和 $E[R]$ 都已经确定。

- **实现细节**

为了在一个集合中所有大小为 k 的分段上进行迭代，需要一个枚举方法。我们要实现的函数是 `all_subsets`，采用迭代器的描述形式。Python 的迭代器不使用 `return` 语句而使用 `yield` 语句返回每个结果，这样能不中断迭代器的执行。

函数 `all_subsets(n, k)` 会枚举基数 k 的所有分段 $S \subseteq \{0,\cdots, n-1\}$。如果 i 是 S 中的最大值，那么对于一个基数 k-1 的分段 $S' \subseteq \{0,\cdots, n-1\}$，S 可以记作 $S' \cup \{i\}$。函数 `all_sebsets` 的实现将使用这一拆分方法。

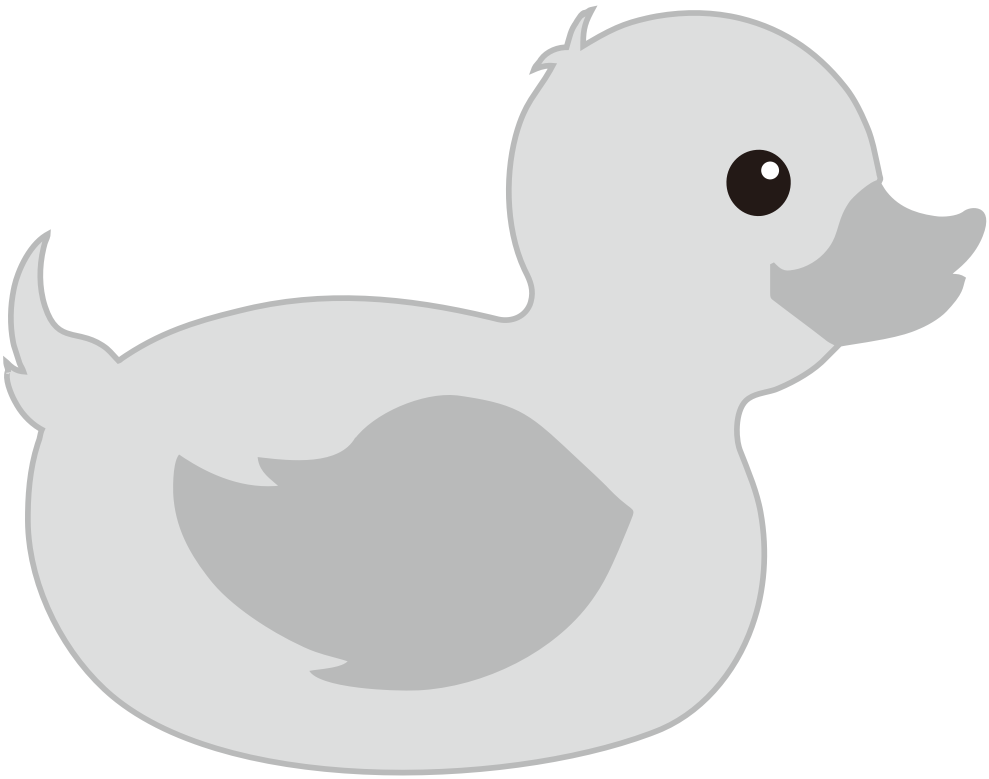
```
def all_subsets(n, card):
    if card == 0:
        yield 0
    else:
        for i in range(card - 1, n):
            for e in all_subsets(i, card - 1):
                yield e | (1 << i)

def arithm_expr_target(x, target):
    n = len(x)
    expr = {}
    for i in range(n):
        expr[1 << i] = {x[i]: str(x[i])}
    tout = (1 << n) - 1
    for card in range(2, n + 1):
        for S in all_subsets(n, card):
            expr[S] = {}
            for L in range(1, S):
                if L & S == L:
```

```
                R = S ^ L
                for vL in expr[L]:
                    for vR in expr[R]:
                    eL = expr[L][vL]
                    eR = expr[R][vR]
                    expr[S][vL] = eL
                    expr[S][vL + vR] = "(%s+%s)" % (eL, eR)
                    expr[S][vL - vR] = "(%s-%s)" % (eL, eR)
                    expr[S][vL * vR] = "(%s*%s)" % (eL, eR)
                    if vR != 0 and vL % vR == 0:
                       expr[S][vL // vR] = "(%s/%s)" % (eL, eR)
# 查找距离目标最近的算式
for dist in range(target + 1):
    for sign in[-1, +1]:
        val = target + sign * dist
        if val in expr[tout]:
            return "%s=%i" % (expr[tout][val], val)
# 如果 x 中包含在 0 和目标值之间的数字，这一部分永远不会执行
pass
```

调试工具

如果你在解决问题的过程中被卡住，不妨和这只鸭子聊一聊，跟它详细、准确地解释你的方案，向它讲解你的每一行代码，这样肯定能帮你找到错误或解决办法。



参考文献

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice Hall, 1993.

[2] Helmut Alt, Norbert Blum, Kurt Mehlhorn, and Markus Paul. Computing a maximum cardinality matching in a bipartite graph in time O($n^{1.5}\sqrt{m/\log n}$). *Information Processing Letters*, 37(4): 237–240, 1991.

[3] Bengt Aspvall, Michael F Plass, and Robert EndreTarjan.Alinear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3): 121–123, 1979.

[4] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Algorithmique*. Dunod, 2010.

[5] Jack Edmonds and Ellis L Johnson. Matching, Euler tours and the chinese postman. *Mathematical programming*, 5(1) :88–124, 1973.

[6] Peter M Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3) :327–336, 1994.

[7] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM* (*JACM*), 34(3) :596–615, 1987.

[8] Rūsinš Freivalds. Fast probabilistic algorithms. In *Mathematical Foundations of Computer Science 1979*, pages 57–69. Springer, 1979.

[9] Harold N Gabow. An efficient implementation of Edmonds' algorithm for maximum matching on graphs. *Journal of the ACM* (*JACM*), 23(2) :221–234, 1976.

[10] Anka Gajentaan and Mark H Overmars. On a class of $O(n^2)$ problems in computational geometry. *Computational geometry*, 5(3) :165–185, 1995.

[11] David Gale and Lloyd S Shapley. College admissions and the stability of marriage. *American mathematical monthly*, pages 9–15, 1962.

[12] Andrew V Goldberg and Satish Rao. Beyond the flow decomposition barrier. *Journal of the ACM* (*JACM*), 45(5) :783–797, 1998.

[13] Carl Hierholzer and Chr. Wiener. Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen*, 6(1) :30–32, 1873.

[14] John Hopcroft and Robert Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6) :372–378, 1973.

[15] T.C. Hu and M.T. Shing. Computation of matrix chain products. part ii. *SIAM Journal on Computing*, 13(2) :228–251, 1984.

[16] Richard M Karp. A characterization of the minimum cycle mean in a digraph. *Discrete mathematics*, 23(3) :309–311, 1978.

[17] RichardM. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2) :249–260, March 1987.

[18] Donald E Knuth. Dancing links. *arXiv preprint cs/0011047*, 2000.

[19] Donald E Knuth, JamesH Morris, Jr, and Vaughan R Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2) :323–350, 1977.

[20] S Rao Kosaraju. Fast parallel processing array algorithms for some graph problems (preliminary version). In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 231–236. ACM, 1979.

[21] Dexter C. Kozen. *The Design and Analysis of Algorithms*. Springer Verlag, 1992.

[22] Chi-Yuan Lo, Jiří Matoušek, andWilliam Steiger. Algorithms for ham-sandwich cuts. *Discrete & Computational Geometry,* 11(1) :433–452, 1994.

[23] Glenn Manacher. A new linear-time on-line algorithm for finding the smallest initial palindrome of a string. *Journal of the ACM* (*JACM*), 22(3) :346–351, 1975.

[24] Sylvain Perifel. *Complexité algorithmique*. Ellipses, 2014.

[25] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *Journal of the ACM* (*JACM*), 44(4):585–591, 1997.

[26] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4) :354–356, 1969.

[27] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2) :146–160, 1972.

[28] Esko Ukkonen. Finding approximate patterns in strings. *Journal of algorithms*, 6(1) :132–137, 1985.

[29] I-Hsuan Yang, Chien-Pin Huang, and Kun-Mao Chao. A fast algorithm for computing a longest common increasing subsequence. *Information Processing Letters*, 93(5) :249–253, 2005.

版权声明



本书是一本基于Python的算法与编程学习书，旨在针对编程竞赛和考试探讨如何优化算法时间复杂度，提高编程的效率。书中详细阐述了经典与特殊算法的实现、应用技巧和复杂度验证过程。本书内容由浅入深，代码精简易读，能帮助读者快速掌握复杂度适当、正确率高的高效编程方法以及自检、自测技巧，是参加ACM/ICPC、Google Code Jam等国际编程竞赛、备战编程考试、提高编程效率、优化编程方法的参考书目。

- **透彻讲解基于Python的高效算法思路与编程要点**
- **战胜编程竞赛技术难关**
- **在线提供更多趣题和拓展实战例子**
- **适合算法爱好者，以及参加编程竞赛和考试的学生和编程人员**

图灵社区：iTuring.cn
反馈/投稿/推荐邮箱：contact@turingbook.com
读者热线：(010) 51095186-600
分类建议 算法 / Python编程
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-48085-9
9 787115 480859 >
ISBN 978-7-115-48085-9
定价：55.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks